2010

Development of a platform for simulating and optimizing thermoelectric energy systems

John Kreuder

Development of a Platform for Simulating and Optimizing Thermoelectric Energy Systems

John J. Kreuder

A Thesis Submitted in Partial Fulfillment of the Requirements for a Master of Science Degree in Mechanical Engineering

Approved by:

Dr. Robert Stevens – Thesis Advisor _____
Department of Mechanical Engineering

Dr. Margaret Bailey – Professor _____
Department of Mechanical Engineering

Dr. Marca Lam – Professor _____
Department of Mechanical Engineering

Dr. Wayne Walter – Department Representative _____
Department of Mechanical Engineering

Date of Approval: 11/15/2010

Department of Mechanical Engineering
Rochester Institute of Technology
Rochester, New York 14623
November 2010

PERMISSION TO REPRODUCE THESIS

Development of a Platform for Simulating and Optimizing Thermoelectric Energy Systems

I, JOHN J. KREUDER, hereby grant permission to the Wallace Memorial Library of Rochester Institute of Technology to reproduce my thesis in the whole or part. Any reproduction will not be for commercial use or profit.

Date: _____

Signature: _____

November 2010

Abstract

Thermoelectrics are solid state devices that can convert thermal energy directly into electrical energy. They have historically been used only in niche applications because of their relatively low efficiencies. With the advent of nanotechnology and improved manufacturing processes thermoelectric materials have become less costly and more efficient As next generation thermoelectric materials become available there is a need for industries to quickly and cost effectively seek out feasible applications for thermoelectric heat recovery platforms. Determining the technical and economic feasibility of such systems requires a model that predicts performance at the system level. Current models focus on specific system applications or neglect the rest of the system altogether, focusing on only module design and not an entire energy system. To assist in screening and optimizing entire energy systems using thermoelectrics, a novel software tool, Thermoelectric Power System Simulator (TEPSS), is developed for system level simulation and optimization of heat recovery systems. The platform is designed for use with a generic energy system so that most types of thermoelectric heat recovery applications can be modeled.

TEPSS is based on object-oriented programming in MATLAB®. A modular, shell based architecture is developed to carry out concept generation, system simulation and optimization. Systems are defined according to the components and interconnectivity specified by the user. An iterative solution process based on Newton's Method is employed to determine the system's steady state so that an objective function representing the cost of the system can be evaluated at the operating point. An optimization algorithm from MATLAB's Optimization Toolbox uses sequential quadratic programming to minimize this objective function with respect to a set of user specified design variables and constraints. During this iterative process many independent system simulations are executed and the optimal operating condition of the system is determined.

A comprehensive guide to using the software platform is included. TEPSS is intended to be expandable so that users can add new types of components and implement component models with an adequate degree of complexity for a required application. Special steps are taken to ensure that the system of nonlinear algebraic equations presented in the system engineering model is square and that all equations are independent. In addition, the third party program FluidProp is leveraged to allow for simulations of systems with a range of fluids. Sequential unconstrained minimization techniques are used to prevent physical variables like pressure and temperature from trending to infinity during optimization.

Two case studies are performed to verify and demonstrate the simulation and optimization routines employed by TEPSS. The first is of a simple combined cycle in which the size of the heat exchanger and fuel rate are optimized. The second case study is the optimization of geometric parameters of a thermoelectric heat recovery platform in a regenerative Brayton Cycle. A basic package of components and interconnections are verified and provided as well.

Acknowledgements

I would like to extend special thanks to my advisor Dr. Robert Stevens for his guidance and problem solving assistance throughout the duration of this project. Additionally, I'd like to thank Dr. Steven Weinstein and committee members Dr. Marca Lam and Dr. Margaret Bailey for their assistance and problem solving expertise.

The project was funded by the New York State Energy Research and Development Authority, whom I'd like to thank for their financial support along with the Rochester Institute of Technology, the Kate Gleason College of Engineering and its Department of Mechanical Engineering.

Finally, to my parents, Mark and Jeanne Kreuder who have supported me throughout college in all of my endeavors – Thank You.

Contents

List of Figures

List of Tables

Nomenclature

Latin Symbol	Parameter
k	Thermal Conductivity [w/(m-K)]
ZT	Thermoelectric Figure of Merit
T	Absolute Temperature [K]
$[J]$	Jacobian Matrix
x_i	i^{th} Guess Vector
$f(x)$	Objective Function
$c(x)$	Inequality/Side Constraint
$ceq(x)$	Equality Constraint
$L(x, \lambda)$	Lagrangian Objective Function
X	Vector of design variables

Greek Symbol	Parameter
ρ	Electrical Resistivity [Ω-m]
σ	Electrical Conductivity [S/m]
Δ	Change in Value
Λ	Lagrange Multiplier

Quantities used herein are all given in SI units: kilogram (kg), meter (m), second (s) Kelvin (K), Volt (V) coulomb (c), radian (-), United States Dollars ($) and their combinations unless otherwise noted.

Computer code is displayed according to MATLAB® syntax.

Vector quantities are stated as column vectors unless otherwise noted.

Variables are in *italics*.

Chapter 1

Overview

The development and study of thermoelectric (TE) modules has been an area of scientific interest for much of the last century [1]. When used to recover waste heat, these devices can potentially boost the thermal efficiency of an energy system. This application is becoming especially relevant as device manufacturing costs fall and better thermoelectric materials are developed. A great deal of effort has gone into module performance modeling and modeling of waste heat recovery platforms [2-4]. However, these models often focus only on the performance of the thermoelectric device and not on the performance of the energy system as a whole. Thermoelectric heat recovery platforms are typically integrated into existing energy systems to recover waste heat. Maximizing thermoelectric power output with respect to cost does not guarantee an optimal configuration at the system level. A system level model could be far more useful than a model that focuses only on the performance of the thermoelectric portion of the system. To determine and analyze the performance of such systems, the Thermoelectric Power System Simulator (TEPSS) software platform is developed and demonstrated herein.

1.1 Background

The Seebeck effect, first discovered in the early 1800s by Thomas Johannes Seebeck is an electrical phenomenon observed when a temperature difference exists across the junction of two dissimilar metals. As long as the temperature gradient persists, a voltage will be present across the junction. This effect can be harnessed for one of two purposes: to generate electricity or to move heat. The Seebeck effect can be harnessed to generate electricity from a number of thermoelectric junctions arranged thermally in parallel and electrically in series. If current is passed through the same circuit of junctions, heat will be pumped from one side of the device to the other, creating a

temperature difference across the device. The latter application is referred to as the Peltier effect.

Thermoelectric generators (TEGs) are comprised of two semiconductor materials; one material is a p-type semiconductor and the other an n-type. Figure 1.1 shows a pair of thermoelectric legs arranged thermally in parallel and electrically in series. Hot and cold side temperatures are labeled T_h and T_c respectively, semiconductor dimensions are labeled for the p and n- type legs (L_p, W_p, L_n, W_n), current is labeled (I) and the load resistance R_o is shown. This configuration is known as a thermocouple. Arrays of thermocouples are assembled together to make thermoelectric modules as shown in Figure 1.2. Thermoelectric modules are mass produced devices that can be used to recover waste heat from thermal processes. Applications for thermoelectric power generation from the Seebeck effect are emerging as manufacturing processes improve and advances in material science are made.

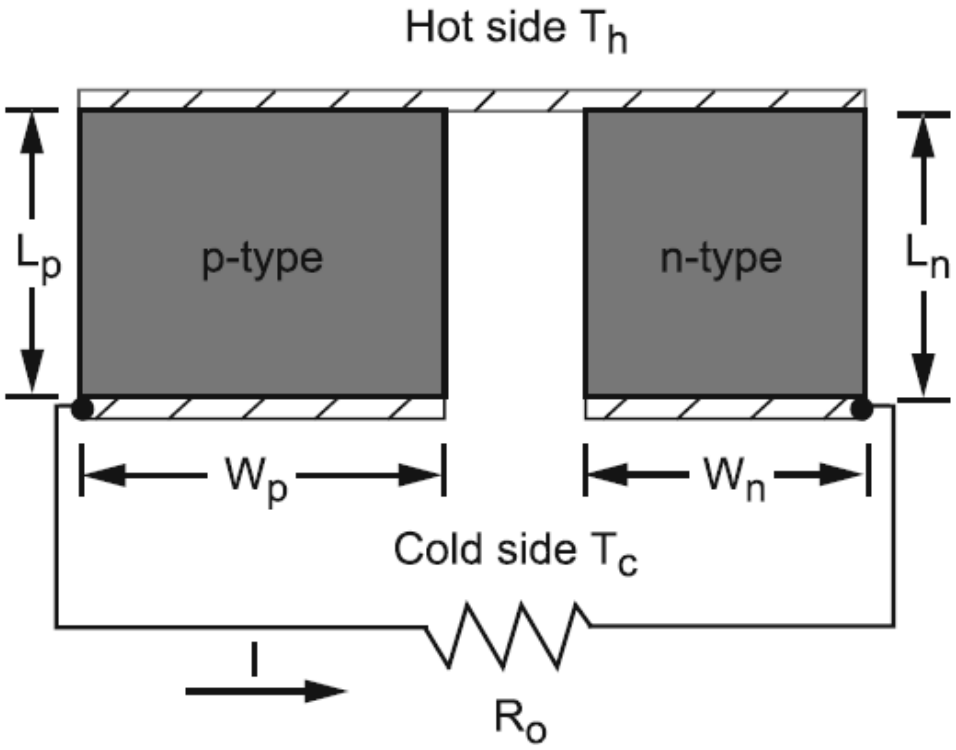

Figure 1.1: A Thermocouple for Thermoelectric Power Generation [5].

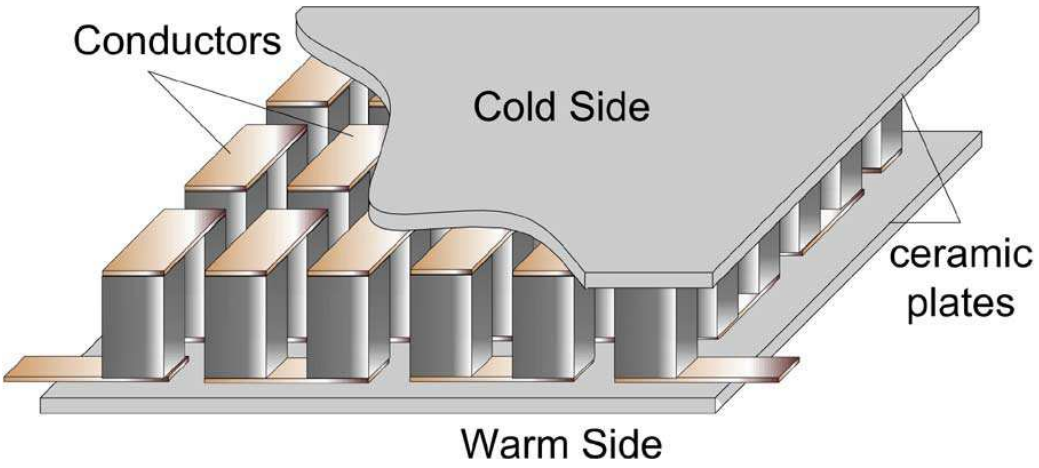

Figure 1.2: Schematic Diagram of a Thermoelectric Module [6].

1.2 Motivation

Energy systems that dissipate waste heat are ubiquitous in modern society. These systems are often highly inefficient when used to produce work and electricity, dissipating fifty to seventy five percent of the available energy as waste heat. The majority of these systems utilize finite resources like natural gas and coal as fuel. In the United States alone, about 50 quadrillion BTU's are dissipated as waste heat annually [7]. If just a small fraction of this heat could be recovered, a huge amount of energy and finite resources could be saved. Thermoelectric device technology has the ability to recuperate a portion of this huge energy resource and convert it into useful electricity. Systems have been devised to recover waste heat from engine exhaust, power plant waste heat streams and for small scale applications such as a projector lamp [1]. Space heating systems and industrial incinerators have also been targeted as potential applications for thermoelectric heat recovery platforms [8,9].

As the cost of thermoelectric technology continues to fall and better materials are engineered, thermoelectric devices are becoming a potentially feasible solution for reducing consumption of finite sources of energy. Improving the way thermal systems utilize energy will reduce the environmental impact associated with the use of most traditional energy sources.

Some modeling has been done to predict the performance of thermoelectric heat recovery platforms in various types of thermal systems. Bethancourt et. al developed a

model for thermoelectric heat recovery for a cross flow heat exchanger [3], while others Hendricks and Lustbader did similar work aimed toward automotive applications [10]. Crane and Jackson's research models predict TE heat recovery from an exhaust gas stream [2]. Much of the modeling is specific to very narrow applications of the technology and no tool exists that is both reliable and versatile for TE power system feasibility analysis and optimization. As the prospect of TE technology becomes more attractive, such a tool would save time and resources with the ability to quickly predict performance and costs of incorporating TE generation into a variety of thermal systems.

As thermoelectric materials improve, new applications for heat recovery become economically feasible. There needs to be a quick and cost effective way of searching for applications of thermoelectric technology. The modeling must be done at the system level to guarantee the economic feasibility of the entire system. A software tool that can be used to quickly simulate and optimize a general energy system is ideal for this purpose.

In much of the literature on thermoelectric power system optimization, the focus is put on maximizing the efficiency of the thermoelectric modules. This is not always the ideal approach, because design engineers are interested in the economic feasibility of the system as a whole, not just the thermoelectric device. Measures taken to boost thermoelectric efficiency can diminish overall system performance. Since the waste heat used for thermoelectric generation is free, it may be more desirable to operate the thermoelectric portion of the system at a lower efficiency in favor of better overall system performance. Optimizing the thermodynamic performance of a system will lead to increased efficiency, but not necessarily the most feasible configuration. A more useful metric of system performance can be obtained by using a thermoeconomic approach. This approach deals more with the trade-offs between costs (capital, maintenance, fuel, etc.) and the thermal performance of the system. When combined with an optimization routine, the thermoeconomic approach to optimizing an energy system is most useful for maximizing the benefit obtained from a system while minimizing overall cost [11].

As thermoelectric device models continue to mature and more versatile heat recovery platform models are developed, a versatile system level modeling tool is needed to incorporate the TEG platforms into applications. Grekas and Frangopoulos [12] develop a self synthesizing energy system based on graph theory in C++. The approach

that is described is exclusively applicable to systems in which the dependent variables relate to a fluid system. Finding the values of the dependent variables defines the thermodynamic state of the fluid system. The model treats mass and energy flows within a thermohydraulic system as dependent variables. User-specified component parameters are treated as independent variables. The dependent variables are solved as functions of the independents recursively, until an optimal set of independent variables is found. If this approach were to be extended to apply to mechanical, electrical and other energy interactions, it could be a useful tool for modeling thermoelectric heat recovery systems.

From the articles available in the literature, it is apparent that the system level models that exist are exclusive to narrow applications. A software platform that could be used to simulate any type of energy system with or without thermoelectric heat recovery would be valuable in determining the feasibility of such a system. Software developed for this purpose would need to execute simulations at the system level, reporting outputs for all of the energy system's components.

1.3 Objectives

The main objective of this research is to develop the architecture for a software tool capable of simulating and optimizing a generic energy system. Energy systems are defined in this context as a complete set of component engineering models, component arrangement information, initial component parameters, boundary conditions, and an initial guess for the operating point. The set of dependent variables within the system will be solved for using the equations that describe the physical behavior of each component in the system. An optimization algorithm will be employed to search for and locate the optimal system configuration with respect to the specified design variables.

The software tool will be capable of simulation with components that contain engineering models of varying complexity. The components may contain simple equations derived from first principles, empirical models or complicated finite element models. The software tool, referred to herein as the Thermoelectric Power System Simulator (TEPSS) is designed to fill the needs expressed in Section 1.2, specifically focusing on simulating and optimizing systems that contain thermoelectric heat recovery

platforms. TEPSS will be designed to be open source so that future users can expand upon the tool to fit their specific modeling needs.

When finished, the tool will be made freely available to the thermoelectric community to help search for feasible applications of heat recovery platforms. A comprehensive user manual will be provided as part of the TEPSS distribution package. The manual will provide the information necessary to arrange the base package of components and domains into a solvable system and run an optimization on one or more component parameters. The user will likely wish to implement more specific engineering models than the ones provided with the base package, so the manual will contain all of the necessary information needed to expand upon the base package.

Two separate case studies will be performed to validate the functionality of TEPSS. The first case study will focus on verifying that the simulation shell is capable of solving the system of equations (engineering model) for an energy system. The system will have multiple domains, phase changes in the fluid domain and both open and closed loops. The second case study will focus on validating the optimization algorithm and it will contain a heat exchanger that utilizes a thermoelectric heat recovery platform as a component. Parameters of the heat recovery platform will be optimized and the results will be analyzed. Conclusions about the technical and economic feasibility will be drawn.

Chapter 2

Introduction

As discussed in Chapter 1, a main goal of this project is to develop a software tool (TEPSS) capable of determining the feasibility of a general energy system with an integrated thermoelectric heat recovery platform. The object-oriented programming approach is assessed as a possible format for TEPSS architecture. Various approaches for simulating and optimizing energy systems are discussed as well.

2.1 Thermoelectric Modules

The phenomenon of solid state conversion of heat into electricity was first observed and studied in the 19th century first by Thomas Johann Seebeck circa 1822 and also by Jean Charles Peltier a decade later. They observed that adding heat to metallic substrates could produce electrical current. William Thompson was the first scientist to fully understand the Seebeck and Peltier effects as they would come to be known [7]. The Seebeck effect is observed when heat passes through a junction of two dissimilar metals, creating a voltage. The Peltier Effect is observed when electric current is passed through a junction of dissimilar materials causing heat to be either absorbed or emitted by the thermoelectric couple.

Thermoelectric device performance is characterized by the dimensionless figure of merit (ZT). This unitless parameter is used to quantify ideal performance for a particular material. The figure of merit can not be directly used to predict module behavior, but a higher figure of merit is an indicator of better module performance and higher efficiency. Certain module level effects such as electrical contact resistance are not accounted for by the figure of merit. Thus it is only an indicator of ideal performance. The figure of merit for a material is defined as:

$$ZT = \frac{\alpha^2 \sigma}{k} T \tag{1.1}$$

Increasing the Seebeck coefficient difference (α) between the materials used will greatly increase the figure of merit at a given temperature. Increasing electrical conductivity (σ) will do the same by reducing the amount of Joule heating taking place within the module. Decreasing the thermal conductivity (k) of the leg pairs will reduce the thermal conductive losses. To achieve non-dimensionality, the figure of merit is multiplied by absolute temperature (T). Current commercially available modules have ZT values of about 0.8 at a temperature of 200°C, while recent advances in nanotechnology have produced ZT values as high as 2.5 in the laboratory [13].

A simple model for thermoelectric material efficiency can be expressed in terms of the Carnot efficiency and the average ZT within the module as shown in eqs (1.2 & 1.3). Where T_H and T_C are the respective absolute temperatures of the hot and cold sides of the semiconductor leg and ZT_H and ZT_C are the figures of merit evaluated at those temperatures [7].

$$\eta = \frac{T_H - T_C}{T_H} \times \frac{M - 1}{M + T_H / T_C} \tag{1.2}$$

where η is efficiency, the first term represents Carnot efficiency and the second term accounts for irreversibilities wherein M is defined as

$$M = \sqrt{1 + \frac{1}{2}(ZT_H + ZT_C)} \tag{1.3}$$

Linking multiple thermocouples electrically in series and thermally in parallel creates a thermoelectric module. In general, one side of each thermocouple is a p-type semiconductor and the other is an n-type semiconductor with a large difference in Seebeck coefficients. The voltage produced by the module is proportional to the temperature difference between the hot and cold sides of the device. The constant of proportionality under open circuit conditions is the difference in the Seebeck coefficients (α) of the two semiconducting materials. The potential to generate power will persist as long as a temperature difference is maintained across the device. This typically requires thermoelectric heat recovery platforms to be part of a heat exchanger because such a device would maintain a temperature difference. An example of such a platform is shown in Figure 2.1.

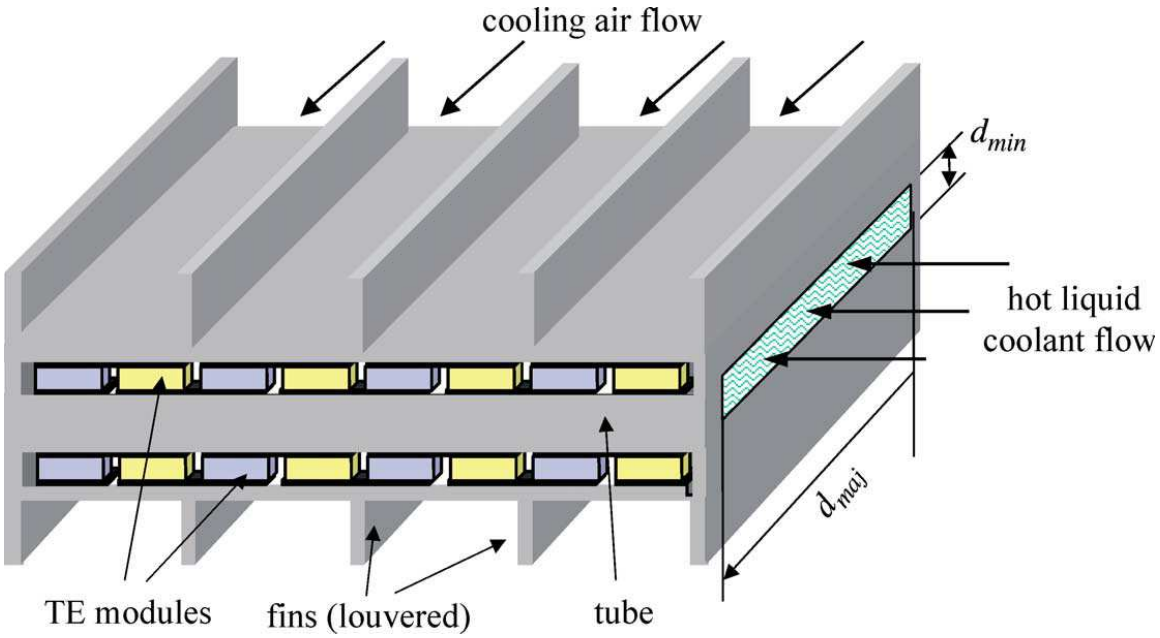

Figure 2.1: Thermoelectric Heat Recovery Platform [2].

From the late 1800's until the mid-1900's little attention was paid to the Seebeck effect as a means of producing electricity. In the 1950s the first application of thermoelectric generators were realized, thanks in part to significant technological advances in materials science and semiconductor theory. Thermoelectric generators were employed to power the American deep space probe Voyager and parts of the Apollo spacecraft [14] because they were robust and scalable. Incremental improvements in materials were made from 1960 until the turn of the century. Since about 2000, there have been significant advancements in thermoelectric materials due to an improved understanding of thermal transport in nanostructures and advances in nanomaterial processing. These advances along with the demand for alternative energy sources have sparked resurgence in thermoelectric research and application development. Currently, most applications are in niche areas such as powering remote sensors and remote external power. The niche applications typically use traditional materials which are commercially available. As next generation materials emerge, there is interest in trying to incorporate thermoelectrics into broader applications such as automotive waste heat recovery, industrial waste heat recovery and microcircuit cooling.

The most common type of semiconductor used in thermoelectric modules is Antimony doped Bismuth Telluride. BiTe modules are mass produced in rectangular sizes measuring a few centimeters each in length and width. Module thickness is typically in the range of a few millimeters. Thermal stresses induced by large temperature gradients across the devices have prevented the sizes of individual modules from being scaled up beyond current sizes, but modules can be connected to one another electrically for large applications.

2.2 Object-Oriented Programming Languages

Object-oriented programming began to emerge in the 1960s as a modularized way to manage increasingly complex software. In contrast with traditional linear programming techniques, object-oriented programming utilizes classes of variables. An instance of a class is called an object, which contains data structures (properties) and a set of functions (methods). Class definitions are reusable pieces of code that define the properties and methods of an object. This modular framework can allow for objects that are self sufficient, containing all of the necessary data and algorithms to operate on themselves.

An object's properties consist of data stored within the object that can be called by any method within the object. Data of any type – even other objects – may be stored as properties within an object. An object's methods are a set of functions that can receive and process data from outside the object as well as data stored within the object. Energy systems can be thought of as a collection of component objects connected together in a particular arrangement to produce a set of outputs with respect to a set of system boundary conditions. The components making up the system behave as independent objects whose throughputs can be modeled independent of the other components in the system or their arrangement. For this reason, components can easily be modeled as objects.

Another useful quality of the object-oriented approach is that classes can inherit the properties and methods of a parent class. This could allow for a master class with all of the required properties and methods to be developed and for certain subclasses to

inherit that data structure, saving the time it would take to develop each subclass in parallel. In a superclass, properties and methods common to all subclasses can be defined once for all subclasses that inherit the class.

The modular nature of objects can be used to prevent accidental data corruption. Properties within an object can be made private so that only the methods within an object are capable of overwriting the property, or the property can be made constant, so that no operation can change its value (read-only).

The architecture of object-oriented programming languages also lends itself to a shell based system in which one object exist as a property within another object, thereby creating a tiered hierarchy of functions that can be executed iteratively from the tiers above. Since the TEPSS architecture seeks to optimize an energy system, it will need to iteratively simulate the system. The ability to repeatedly call the simulation using a shell based system will be a key feature in the architecture of TEPSS. A diagram of the proposed data flow structure for TEPSS is presented in Figure 2.2. This data flow structure is described in great detail in Section 3.3.

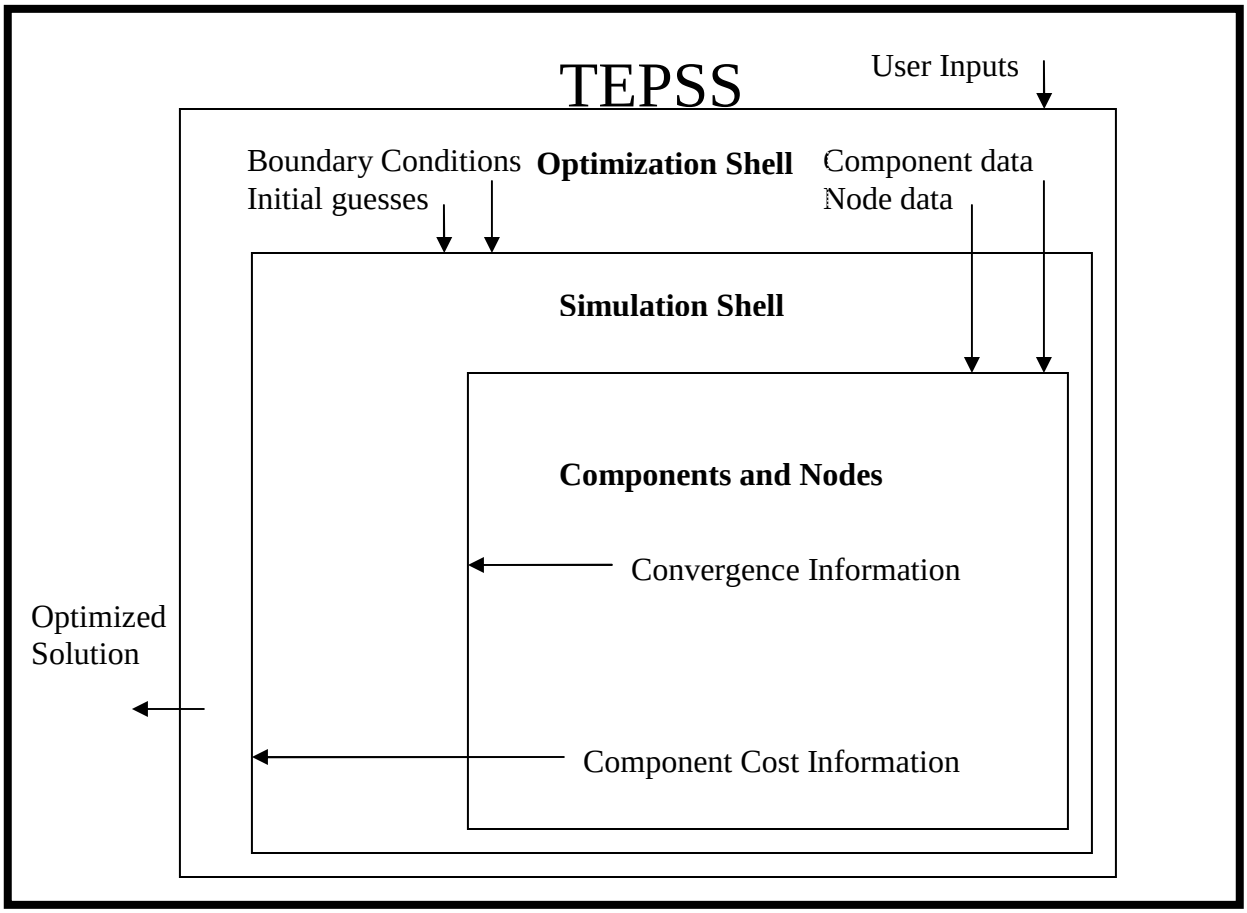

Figure 2.2: Proposed Data Flow Structure for TEPSS

There are over 100 object-oriented programming languages. The most well known of which include C++, C#, Fortran, Java, and Objective – C [15]. MATLAB® is a software tool used widely by engineers for modeling and optimization. Beginning in 2008, MATLAB introduced object-oriented programming into its software package. While the MATLAB coding environment is uncompiled, the option exists within the MATLAB software to compile the TEPSS algorithms into an executable. The combination of a readily available optimization package and an object-oriented programming software package makes MATLAB/Simulink an excellent software platform for the development of TEPSS.

Many engineers are unfamiliar with object-oriented programming. TEPSS will be developed to take advantage of the modularity of the object-oriented paradigm, while keeping user inputs from requiring extensive programming knowledge.

The biggest difference between object-oriented and traditional linear programming techniques is the widespread use of data structures. While MATLAB has historically focused on matrix and array based nomenclature, the addition of object-oriented programming to MATLAB in 2008 brought with it the addition of structure-based nomenclature. While array notation remains unchanged in MATLAB from previous versions, the user may also choose to store data in the form of a structure.

Conceptually, structures are variables that store other variables. A structure is a variable in which fields exist. Each field stores another variable. Table 2.1 contains an example of structural notation for a structure named *struc* and its three fields *f1, f2* and *f3.*

Table 2.1: Using Structure Based Nomenclature

Structure	**Field**	**Accessed as**
struc	*f1*	*struc.f1*
	f2	*struc.f2*
	f3	*struc.f3*

The values of the fields of the structure are accessed by placing a period followed by the field name after the name of the structure as shown in the third column in Table 2.1 above. Structural notation is useful for storing different data types since each field is a unique variable. This is a major advantage over arrays. Fields of a structure may be of any data type, including arrays, cell arrays, strings and character arrays, objects and other structures. If a structure is stored in a field of another structure, then it too will have fields, creating a hierarchy of fields within a single structure. This concept is illustrated in Table 2.2, where the variable *struc* has three fields, one of which, *size,* is a structure, creating the tiered structure shown.

Table 2.2: Nested Structures

Structure	Field	Data Type of Field	Subfield	Value	Accessed as
struc	*size*	structure	*Length*	*1*	*struc.size.length*
-	-	-	*Width*	*pi()*	*struc.size.width*
-	-	-	*Height*	*4/5*	*struc.size.height*
-	*mass*	array	-	*[3]*	*struc.mass*
-	*name*	string	-	*'block'*	*struc.name*

Structural notation is also useful because it lends itself to the idea of modularity. If all data pertinent to a specific part of the modeling process is stored within a single structure then it can all be passed into and out of functions by passing only a single variable, the structure. There is no limit to the number of fields or tiers a structure may have. The names of the fields may be whatever the user chooses.

One disadvantage to using structural notation is that users often make typographical errors while storing data in a structure, and these errors can go unnoticed until much later. While such an error would immediately result in an error when using array notation, mistyping the name of a field when attempting to store data in a structure will create a new field parallel to the intended target field.

Objects are similar to structures in that they are capable of storing multiple types of data, so it makes sense that objects use structure-based nomenclature. The properties of an object are homologous to the fields of a structure, and so they can be accessed in a similar manner. For an object *obj* with fields *A*, *B* and *C*, the values of the properties can be accessed as *obj.A*, *obj.B* and *obj.C* respectively. However, objects differ from structures in several key ways. Most importantly, objects contain methods, which are functions capable of operating on the data stored within the object as well as external inputs. Methods of an object can be accessed similar to the way properties are accessed, except inputs are supplied. For example, to access the method *add* in the object *obj*, a method which adds together two scalar inputs and outputs the sum, the user would execute:

[*sum*] = *obj.add*(*input1*, *input2*);

Another major difference between objects and structures is that an object's properties must be declared within the class definition file. This eliminates the aforementioned drawback associated with structure-based nomenclature in which a typographical error can easily go unnoticed. All properties are public by default, which means that property values can be read and written from outside of the object. Properties may also be set to private (can only be written to from within the object) and constant (read only).

The modularity of objects and the hierarchy of structural notation make object-oriented programming an excellent paradigm for the development of TEPSS. Using this scheme, it will be possible to pass large amounts of data of several types within a single variable. Tiered structures will allow for organization of data and easy interpretation through the use of meaningful variable names, which is not easily accomplished with array-based nomenclature.

2.3 Simulating an Energy System

2.3.1 Overview

Energy systems are modeled using the mathematical relationships that relate system behavior to component configuration and boundary conditions. These relationships are often derived from physical laws. Energy system modeling is done particularly often on thermodynamic systems, in which the laws of conservation of mass and energy are key relationships. The mathematical models that are used can be analytical models derived directly from physical laws, empirical models determined from experimental results or finite element models which discretize and simplify complicated equations. Whatever the case, the goal of modeling a system is to predict its performance given a set of inputs.

Energy systems are comprised of individual components. The behavior of each of these components can be described with a set of equations. The equations are often derived from conservation laws or other equations with physical meaning, relating the dependent variables to a set of independent variables and/or boundary conditions.

Linking components together forms an energy system and the variables are shared between two interconnected components. With an appropriate set of boundary conditions and independent variables the system of equations can be solved, yielding the values of the set of dependent variables for the given set of inputs.

The system of equations presented within the component models may not be easily solvable. While square systems of linear equations can be guaranteed to have solutions that are unique, the same can not be said for nonlinear systems of equations. Many physical phenomena are nonlinear, and therefore modeling an energy system is likely to require solution of a system of nonlinear algebraic equations (SNAE). Systems of nonlinear equations are not guaranteed to have a solution, and if a solution exists, it is not guaranteed to be unique. Furthermore, matrix based solution methods can not be used to solve SNAE without first linearizing the equations. After solving the linear system of equations, the solution is checked against the nonlinear system, relinearized and repeated until the two solutions agree. This approach forms the basis for iterative solution of SNAE. Such iterative approaches are useful for solving coupled systems in which some or all of the equations contain more than one dependent variable.

Figure 2.3 illustrates a hypothetical energy system in which mechanical work is produced by a Brayton Cycle. In traditional regenerative Brayton Cycles exhaust heat leaving the turbine is recovered and used to preheat the air before it enters the combustion chamber. The heat exchanger in the hypothetical system functions as a thermoelectric heat recovery platform. A portion of the heat passing through the heat exchanger will be converted directly into electrical power. This DC electricity could be used to power instrumentation within the system, it could be inverted and exported to the power grid or it could be used for some other purpose.

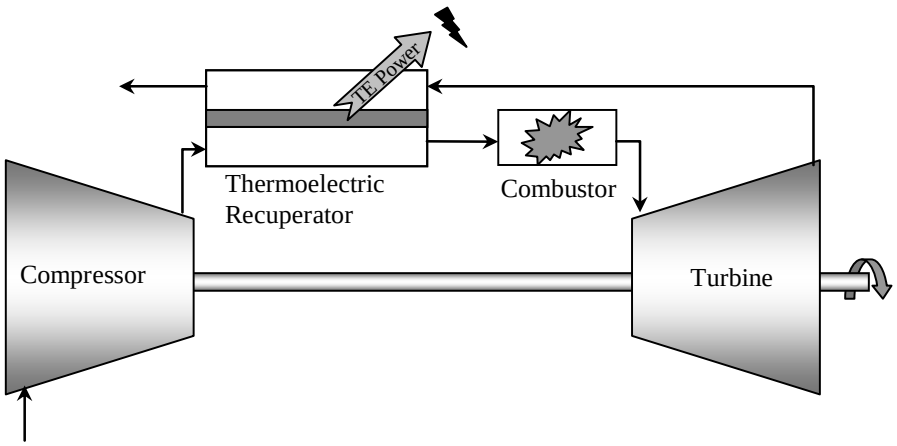

Figure 2.3 Hypothetical Energy System [16].

In such a system, the pressures, temperatures and mass flow rates at each connection depend upon the system boundary conditions and the configuration of each component; therefore they are treated as the dependent variables of the system unless they are fixed by a boundary condition. Conservation laws can be applied to relate the mass and energy values across each component. If these relationships are nonlinear and coupled to one another, then a nonlinear equation solving algorithm should be used to determine a solution. Since TEPSS is required to solve a general system of nonlinear algebraic equations, it will utilize one of these algorithms. Such techniques are discussed in Section 2.3.2.

2.3.2 Equation Solvers

Solving steady state engineering models that do not contain differential equations lies more in the field of linear algebra than it does in the field of system modeling. The energy system simulation process for steady systems essentially boils down to finding a real and possible solution to the *n* coupled linear or nonlinear algebraic equations contained within the components' engineering models. While the goal of finding a solution to a system of nonlinear algebraic equations (SNAE) may sound simple, finding a general and reliable algorithm to do so has proven notoriously difficult throughout history [17]. Since the system contains nonlinear equations, no guarantee exists that a solution exists and if a solution is found, no guarantee exists that the solution is unique. Fortunately, a system of equations that represents a well-posed, steady, physical system

often has only one or a few real solutions and even that set of solutions can often be reduced further by excluding solutions with impossible values like negative absolute pressures and temperatures.

The two main types of approaches for solving SNAE are iterative methods and holomorphic methods [18]. Holomorphic methods can be useful for arriving at multiple geometrically independent solutions of a SNAE, but the techniques are computationally demanding, they involve imaginary and complex paths to reach each solution and the techniques are limited to dense polynomial systems. These restrictions prevent holomorphic methods from applying directly toward a general and sparse SNAE as required by TEPSS. Iterative methods on the other hand can handle any independent SNAE, but they find only one solution at a time and the solutions found are sensitive to the initial guess [17].

The iterative algorithms used to find the root(s) of a nonlinear equation are closely linked to optimization algorithms. The key difference being that root finders are used on a derivative order lower to find the root(s) of an equation rather than to find the root(s) of the first derivative of an equation. The algorithms described by Vanderplaats [19] can be applied to finding a root of a single nonlinear equation using iterative methods. These algorithms include bisection, golden section, Newton's method, secant method and other similar routines. The methods listed use either zero order information or first derivative information and one or more initial guess points to isolate the root(s) of a single linear or nonlinear equation. Convergence is achieved as step size in between iterations approaches zero and so does the value of each equation for which roots are being found.

The golden section and bisection algorithms do not scale easily to multidimensional space. Newton's Method, however, does. In this case, the Jacobian matrix of a system of equations is used in place of the first derivative of a single equation and a linearized system of equations is solved using the iterative approach expressed by the vector eqs (2.1 & 2.2).

$$\Delta x_i = -[J(x_i)]^{-1} \times f(x_i) \qquad x \text{ and } f \text{ are } n\text{-by-1 vectors} \tag{2.1}$$

$$x_{i+1} = x_i + \Delta x_i \tag{2.2}$$

where x_i is the i^{th} guess of a solution of x. $f(x_i)$ is a vector containing the scalar values of all f when evaluated at x_i and $[J(x_i)]$ is the n-by-n Jacobian matrix of all $f(x)$ calculated at the point x_i. The algorithm converges as $f(x_i)$ approaches a zero vector.

Other multidimensional root finding algorithms include conjugate direction methods and evolutionary algorithms such as particle swarm optimization and genetic algorithms [20]. Similar to one dimensional root finders, these algorithms use only zero and first order derivative information. This aspect is preferable for solving a general system of equations because numerical derivatives are used. The algorithm selected for TEPSS should avoid calculating higher derivatives in root finding due to the time required to evaluate each equation at multiple points and the loss of accuracy that comes from calculating higher orders of numerical derivatives from a limited number of data points.

Newton's Method is a candidate for implementation as a simulation algorithm for TEPSS because of its relative simplicity to implement in n dimensions, its quadratic convergence and its reliability given an adequate initial guess. Newton's Method and its numerical adaptations are far and away the most widely implemented methods for solving SNAE. The method can easily be adapted to use a Jacobian matrix containing numerically calculated derivatives, although this may slow the convergence rate. Only the first partial derivatives (Jacobian Matrix) of the SNAE and an initial guess are required for the algorithm to iteratively arrive at a solution. Some of the drawbacks to the Newton's Method are that it has been known to fail when a poor initial guess is provided and that its convergence slows down in the vicinity of root multiplicities [21]. Multidimensional implementations of Newton's method using numerically populated Jacobian matrices are often referred to as quasi-Newton methods.

Implementing Newton's Method in multiple dimensions can involve the introduction of free variables. Each equation is first formulated as $f_i(x) = 0$, where $f_i(x)$ is the i^{th} of n equations in the system. Then a free variable is introduced such that $f_i(x) = e_i$. The value of e_i represents the amount by which $f_i(x)$ is violated when evaluated at x. If $e_i = 0$, then the original equation, $f_i(x) = 0$, is satisfied. The solution method therefore seeks to minimize the norm of the vector r.

He's Method for nonlinear equation solving is a method that leverages Lagrange multipliers to circumvent the typical failure modes of Newton's Method [22]. These failure modes include failure to converge and outright divergence from a solution often due to a poor initial guess. While He's Method is capable of leveraging numerical derivatives to carry out its computations, it is not easily expandable from a one dimensional to an *n* dimensional solution algorithm.

In earlier work Borisevich, Potemkin and Strunkov published a discussion of several algorithms used to solve SNAE. They make mention of a spectral method, a multidimensional resultant method and a method of Groebner basis reduction; all of which reduce the solution process to a problem involving rank 1 matrices in a subspace of matrices constructed specially for the system of equations [17]. The methods do not solve the aforementioned problems regarding the existence and uniqueness of a solution.

The quasi-Newton Method is most suited for use in TEPSS because it is scalable to *n* dimensions and easy to implement, because it converges in a superlinear fashion even when numerical derivatives are used and because it has been widely implemented for similar applications over the past 50 years. As with all SNAE, the existence of a solution for any system of nonlinear equations is not guaranteed. Since TEPSS deals with physical systems, it can be stated that at least one solution should exist so long as the problem is well posed. With regards to the uniqueness of the solution, the quasi-Newton Method needs an initial guess in the vicinity of a root; if a poor guess is given, an unrealistic solution may be located instead. The degree of accuracy required in the initial guess depends on the system of equations. Since most of the systems modeled in TEPSS are based on physical systems, it may be possible to deduce a relatively accurate initial guess from a similar physical system that already exists.

2.3.3 Systems Approach

Tools that have been developed to date for modeling of physical systems tend to focus on dynamic systems and system control theory [23]. While these types of models should be adequate for predicting steady state system performance they are exceedingly complex for the required application. When dealing with coupled systems of nonlinear algebraic equations and numerical derivative information in *n* dimensions, the solution

process can become quite difficult without the addition of the trivial time dependence. In a steady system, the component engineering models do not depend on time.

Simscape, a portion of the MATLAB/Simulink software package, is an environment designed for simulation of dynamic physical systems. Released in 2008, the Simscape environment utilized Simulink solvers to model dynamic interactions between components connected in a block diagram by a series of nodes. In Simscape, the user defines through and across variables within the components (also known as flow and effort variables respectively). These are the variables for which the solver is iteratively solving. The values of these variables are stored on the ports where nodes connect to components. Across variables are constant within a node (and at all adjacent ports) and through variables are instantaneously conserved as they enter and leave a node. Nodes connect to ports of a particular domain on a component and only ports of the same domain can be connected to a given node. In the preliminary program, attempts were made to accomplish many of the goals of TEPSS. However, after testing and analysis, the platform experienced difficulty solving SNAE, primarily because Simscape is based on a system modeling approach where the important effect is the time dependence of the system; a time dependence that does not exist within the SNAE. While the Simscape platform struggled to solve steady systems in the absence of time dependent states, the component, node and domain architecture used in the environment served as a major inspiration for the architecture of the final TEPSS program.

2.4 Optimizing an Energy System

TEPSS is intended not just to model an energy system and produce a measure of feasibility, but also to optimize the system with respect to one or more design variables to maximize or minimize a feasibility metric (objective function). The first optimization algorithm was pioneered by Carl Freidrich Gauss in the early 1800s. Known as ‘steepest descent,’ the method finds the direction in which the objective function is decreasing the fastest and takes a step in that direction [19]. In optimization, an objective function is a scalar value that the optimization algorithm seeks to minimize by adjusting the values of the design variables. Since Gauss, many optimization algorithms have improved on his

procedure, becoming faster and more robust. The resulting set of procedures came to be known as gradient-based optimization methods. A second family of optimization methods known as evolutionary methods was popularized more recently with the availability of large amounts of computing power [24].

Some of the other types of gradient based methods are sequential linear programming, sequential quadratic programming, simplex method, method of Lagrange multipliers, Newton's method and conjugate direction methods. Not all of these methods can directly deal with design variable constraints. Since all gradient-based methods rely on the existence of a derivative to find a search direction and step size, gradient-based methods tend to break down when the objective function and its first derivatives are discontinuous or non-existent. Some evolutionary algorithms, including the popular genetic algorithm, are based loosely on a random search of the design space. They rely on a large sampling of the design space, but they don't require derivative data. While the process may take longer to execute, it is less likely to encounter problems in cases where the objective function is non-smooth.

While gradient based methods are fast and efficient compared to evolutionary methods, they are not always able to find the global minimum of the design space, especially in cases where the objective function is highly non-convex. That is, if a large number of local minima exist within the design space, gradient-based methods are likely to get stuck in one of the local minima, detect convergence, and kick out of the optimization routine. This can happen in non-convex design spaces if the initial guess (starting point) is far from the global minimum. The only way to guarantee a global minimum without a priori information is to try a large sample of starting points. Some evolutionary algorithms, including particle swarm optimization, sample the design space at more than one point at a time. While these routines can become highly computationally intensive, they are more likely to find a global minimum on the first attempt.

Constrained optimization requires use of optimization algorithms that search for minimum of an objective function that lies within the feasible region of the design space. There are three types of constraints that can be levied on a design space: equality constraints, inequality constraints and side constraints. Equality constraints state that a linear or nonlinear combination of two or more independent design variables is

constrained to equal a specific value. Inequality constraints do the same thing, but the combination of design variables is constrained to be less than or equal to a specific value. Side constrains place boundaries on the design space [19]. Equality constraints are traditionally formulated according to the convention $ceq(\bar{x}) = 0$ inequality constraints as $c(\bar{x}) \leq 0$ and side constraints as $c(\bar{x}) \leq 0$. Optimization problems are conventionally posed in the following manner:

$$\min_{x}(f(\bar{x})) \text{ Subject to the set of constraints: } c(\bar{x}) \leq 0 \text{ and } ceq(\bar{x}) = 0 \qquad (2.3)$$

The system is subject to *i* inequality constraints *c*(*x*) and *j-i* equality constraints *ceq*(*x*) for a total of *j* constraints. A solution vector $\bar{x}^*$ exists if all constraints are satisfied and the value of the objective function is non-decreasing in all feasible directions. In gradient based optimization this is sufficient for a global minimum only if the objective function is convex in the design space given. If not, then the only way to guarantee a global minimum is to use more starting points and run the simulation repeatedly.

A stationary point is a point at which all partial derivatives of the objective function are zero. In unconstrained optimization problems, the stationary points of the objective function form a set of possible solutions to the problem. Only those points for which the Hessian is positive definite are true minima. In constrained problems, a solution may exist at a stationary point or at the boundary of a constraint. Often the minima of the objective function do not satisfy all of the constraints, thereby excluding them from the set of possible solutions.

The classical approach to solving constrained optimization problems is The Method of Lagrange Multipliers (MLM). This is done by converting the constrained problem into an unconstrained problem and reformulating the objective function in an unconstrained manner such that minima will only exist in the feasible region. Lagrange multipliers (λ) are introduced into the problem to indicate whether a constraint is violated or satisfied. The Lagrangian optimization equation is stated as [19]:

$$L(x,\lambda) = f(\bar{x}) + \sum_{i=1}^{p}(\lambda_i * c_i(\bar{x})) + \sum_{j=p+1}^{m}(\lambda_j * ceq_j(\bar{x})) \qquad (2.4)$$

where *p* is the number of inequality constraints, *m* is the total number of constraints and *m-p* is the number of equality constraints. λ_i must be non-negative. Possible minima exist

at the stationary points of $L(x, \lambda)$. At a stationary point of $L(x, \lambda)$, the gradient of the original objective function cancels with the gradient of the Lagrangian terms to yield a zero vector. The magnitudes of the Lagrange multipliers (λ) are determined by this requirement.

Lagrange multipliers for equality constraints are calculated as follows: Let x^* be a vector of design variables comprising a solution to the optimization problem. If the gradients of all of the constraint functions are linearly independent and L is a convex function, then a unique vector λ exists such that

$$\nabla L(x^*, \lambda^*) = 0 . \tag{2.5}$$

This statement is true at relative minima, maxima and saddle points of L. To guarantee a minimum it must also be true that a vector λ exists satisfying

$$\nabla_x ceq(\bar{x})_i^T \times \bar{\lambda} = 0 \tag{2.6}$$

for all stated equality constraints $ceq(x)$. At a minimum the Hessian, $\lambda^T \times \nabla^2_{xx} L(x^*, \lambda^*) \times \lambda$ is positive definite with respect to all vectors orthogonal to $ceq(x)$. Simply put, the objective function is non-decreasing in all feasible directions. At this point, the gradient of the i^{th} constraint function $ceq(x)$ and the gradient of the Lagrangian objective function $L(x, \lambda)$ are directly proportional to one another. This constant of proportionality is λ_i. This procedure isolates the stationary points of the Lagrangian objective function, at which a minimum exists if and only if the Hessian matrix of the Lagrangian objective function is positive definite.

The key difference between inequality and equality constraints is that feasible solutions to the optimization problem always lie on the boundary of equality constraints. When dealing with inequality constraints, the solution to the problem may exist on the constraint boundary or in the feasible space on one side of the constraint, but not on the infeasible side. For this reason, a similar but separate approach is used to deal with inequality and side constraints in MLM. Let x^* be a vector of design variables comprising a solution to the optimization problem. If L is convex and the gradients of $c(x)$ are linearly independent, then a solution exists such that $\nabla L(x^*, \lambda^*) = 0$. Where the statements $\lambda^* \geq 0$ and $\lambda_i{}^* c_i(x) = 0$ are true in feasible space. For an active inequality constraint, $c_i(x) = 0$ and for an inactive constraint λ_i equals zero, satisfying the latter

equation. For infeasible values, the Lagrange multiplier λ_i is negative; the magnitude of the multiplier depends on the amount by which the constraint is violated, as with equality constraints.

Constrained optimization adds a considerable amount of complexity to a problem. As an alternative, sequential unconstrained minimization techniques (SUMT) have been developed and employed in the past to circumvent these complexities [19]. SUMT techniques reformulate the original objective function $f(x)$ to create a new pseudo-objective function. This is done by adding together the original objective function and a piecewise penalty function. The penalty function's value is zero for all feasible values and greater than zero for all infeasible values. The magnitude of the penalty function is usually superlinearly proportional to the amount by which the constraint is violated, providing the optimization routine with a descending path into the feasible region. Eq. (2.7) shows how the external penalty function is utilized to form an unconstrained problem:

$$g(x) = f(x) + P \cdot max(0, c(x))^r + P \cdot ceq(x)^r \tag{2.7}$$

where $g(x)$ is the pseudo-objective function being minimized, $f(x)$ is the original objective function and r is a real number greater than 1, typically not exceeding 3. P is on the order of 1 until convergence. Then P is increased and additional iterations of the solution approach are performed to produce a solution of the desired accuracy. The process of ramping up P prevents the gradient of the objective function from rapidly trending to infinity for design variable guesses lying just outside of the feasible region. While ramping up P is not always necessary, it provides a more robust approach than using a large P to start.

Optimization problems sometimes arise where some of the design variables can only take on a discrete set of values, while the rest are still continuous. This class of problem is called a mixed integer problem. These are often solved in two steps. First, solving the problem as though all design variables are continuous and then locking all but the discrete variables in place and searching the nearby design space for the minimum with the discreet design variables taking on only their possible discrete values in the vicinity of the continuous solution. If nonlinearities exist within the objective function or

constraints, then the problem is classified as a mixed integer nonlinear problem (MINLP) [25].

Sequential quadratic programming (SQP) methods are currently the state of the art in gradient-based nonlinear constrained optimization methods. The methods seek to closely mimic Newton's Method for optimization by utilizing second derivative information. A quadratic programming sub-problem is formulated by linearizing the objective function and its constraints. The solution to the sub-problem is the optimal step in each design variable as dictated by the linearized objective function and constraints. The sub-problem takes the form:

$$\min_d f(x) = \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d \tag{2.8}$$

subject to: $\nabla ceq_i(x_k)^T d + ceq_i(x_k) = 0 \qquad i = 1,2,\ldots,p$ and

$$\nabla c_j(x_k)^T d + c_j(x_k) \le 0 \qquad j = 1,2,\ldots q$$

where p and q are the respective number of equality and inequality constraints on the original function. H_k is the approximation of the Hessian matrix of the Lagrangian objective function on the k^{th} major iteration and d is the search direction. The solution is used to form a new iterate x such that:

$$x_{k+1} = x_k + \alpha_k d_k \tag{2.9}$$

where α_k is a scalar that produces the largest decrease in the objective function $f(x)$ as determined by a line search in the direction of the vector d.

A single major iteration of an SQP algorithm consists of three steps:

1) Update the Hessian matrix using one of many available methods.

2) Solve for the direction of steepest descent d_k by solving a QP sub-problem.

3) Perform a line search of $f(x)$ in the direction of d to find the optimal step size scalar α_k.

In SQP solution approaches, the Hessian matrix does not need to be calculated directly at each iteration. On the first iteration the Hessian may be directly calculated or an initial guess may be used (identity matrix is common), but various methods have been developed that update the Hessian in later iterations of x. The Hessian update methods are based on the Hessian's previous value, the change in the gradient of $f(x)$ and the size and direction of the step taken. The most common Hessian update relation is known as the

Boyden–Fletcher–Goldfarb–Shanno (BFGS) method. In this method, the updated Hessian matrix H_{k+1} is calculated according to the relationships in eqs 2.11 & 2.12 [19]:

$$\text{let } y_k = \nabla f(x_k) - \nabla f(x_{k-1}) \tag{2.10}$$

$$H_k = H_{k-1} + \frac{y_k y_k^T}{\alpha_{k-1} y_k^T d_{k-1}} - \frac{H_{k-1} d_{k-1} (H_{k-1} d_{k-1})^T}{d_{k-1}^T H_{k-1} d_{k-1}} \tag{2.11}$$

The linearized sub-problem in eq. (2.8) must then be solved to determine *d*, the direction of steepest descent of the linearized objective function, satisfying the constraints. The Lagrangian of the sub-problem is formulated restating the sub-problem as:

$$d_k = \min_d \left(\frac{1}{2} d^T H_K d + \nabla f(x_k)^T d + \sum_{i=1}^{p} (\lambda_i * ceq_i(x_k^T) d) + \sum_{p+1}^{p+q} (\lambda_j * c_j(x_k^T) d)\right) \tag{2.12}$$

Since this is now a linear system, it can be solved using a host of methods, ultimately arriving at the solution vector *d* for which the linearized Lagrangian function is most rapidly decreasing. The values of the Lagrange multipliers are also determined at the solution *d*. The values of the vector λ are determined in accordance with eq. (2.6) for all equality constraints and $\nabla_x c(\bar{x})_i^T * \bar{\lambda} \le 0$ with $\lambda_i \ge 0$ for all inequality constraints. This solidifies the requirement that the gradient of the original objective function and the gradient of the product of the constraint residuals and their respective Lagrange multipliers cancel out with one another, producing a stationary point in the Lagrangian (eq. (2.4)) at possible minima.

Finally, a one dimensional line search is performed to determine the magnitude of the step size to be taken in the optimal search direction *d*. A line search consists simply of substituting $x = x_k + \alpha_k d_k$ into the original nonlinear objective function where x_k and d_k are known. Minimizing $f(\alpha)$ using a one-dimensional second-derivative test will yield the optimal value of α.

After the line search is completed the values of the design variables are updated using eq. (2.9), the objective function is relinearized and iterations proceed until convergence criteria are met. These include step size magnitude, constraint satisfaction, the change in the objective function value between iterations, change in Lagrange multiplier values and/or changes in the Hessian matrix. At the solution, the vector *x*

represents a minimum of the nonlinear constrained optimization problem posed in eq. (2.8).

Gradient-based optimization methods have been studied thoroughly and collections of optimization software tools are now packaged together and distributed commercially. Among the many of these packages are the MATLAB Optimization Toolbox, Mathematica and NLOpt. Each package contains a set of optimization algorithms that can be leveraged to solve linear or nonlinear optimization problems with or without variable constraints.

MATLAB's Optimization Toolbox contains the *fmincon* function, which uses the SQP procedure described by eqs (2.8 – 2.12). The direction vector *d* is found using one of three internal quadratic programming algorithms.

Chapter 3
Thermoelectric Power System Simulator (TEPSS)

The object-oriented programming environment in MATLAB is used to develop the architecture for simulating and optimizing thermoelectric energy systems. In this chapter a high level overview of the system is given, followed by a detailed discussion of data flow and processing for both the simulation and optimization routines leveraged in TEPSS.

3.1 Architectural Overview

The MATLAB programming environment is selected for development and execution of TEPSS. The reasons for this selection are because it is widely used by engineers both in industry and academia, and because of the availability and completeness of the add-on MATLAB Optimization Toolbox. The Optimization Toolbox contains a host of gradient-based optimization functions. The most versatile of these functions is the *fmincon* algorithm, which can solve linear and nonlinear optimization problems with linear or nonlinear constraints (equality and inequality) within a design space that can be specified. Therefore, the *fmincon* algorithm will be used to solve the optimization problem presented within TEPSS. An adaptation of Newton's Method will be implemented to determine the system operating state by solving the set of equations put forth by the component engineering models.

As discussed in Section 2.2, the object-oriented programming approach allows variables to be assigned to a class. Objects, instances of a class, are variables that have the set of properties and methods specified in the class definition. The methods are functions that can manipulate data stored within the object's properties or data passed in from outside of the object. The properties of an object consist of fields containing other

variables that are accessible to the methods of the object. A property field may store data of any type, including other objects.

The modular design of objects and the ability to store objects within other objects are qualities of object-oriented programming in MATLAB that are useful for development of TEPSS. The modular nature of energy system components can easily be mimicked by the use of objects to represent energy system components and the nodes that interconnect them (Section 2.2). The proposed configuration of TEPSS involves an optimization algorithm working over the top of a simulation routine. This shelled architecture can be imitated using object-oriented programming by storing objects representing each shell within the property fields of the object representing the umbrella shell. These aspects of object-oriented programming will allow the platform environment to effectively emulate the physical structure and interactions of energy systems. Refer back to Figure 2.2 for an illustration of the proposed shell architecture and data flow.

In TEPSS, component objects are connected to one another by node objects. Node objects (nodes) store the dependent variables which are being solved during simulation. The variables stored on a node depend on the domain to which the node is assigned by the user. Components connected to the same node share these node variables, creating the interconnection. Node and domain design is discussed in full detail in Section 4.3.

Figure 3.1 shows a hypothetical system that graphically illustrates the relationship that exists between component and node objects. Each component can access only the node variables of nodes attached to the component. For example, nodes 1 and 2 are both *fluid* nodes for which there are three node variables (mass flow rate, specific enthalpy and absolute pressure). Consider the statement that the rate of mass flow entering component 1 at node 1 will equal the rate of mass flow leaving component 1 at node 2 under steady state conditions. This relationship would likely be reflected in the engineering model of component 1 through the expression *massflow_in* – *massflow_out* = 0; where *massflow_in* and *massflow_out* hold the corresponding nodal mass flow values. Similar statements in the engineering model of component 1 will describe the physical relationships between the values of the other node variables (enthalpy and pressure) between nodes 1 and 2 in terms of the component's parameters. The developer of the component may choose to use any sort of model to describe the changes in mass flow, enthalpy and pressure across or through

component 1, ranging from a very simple relationship to a computationally intensive finite element model.

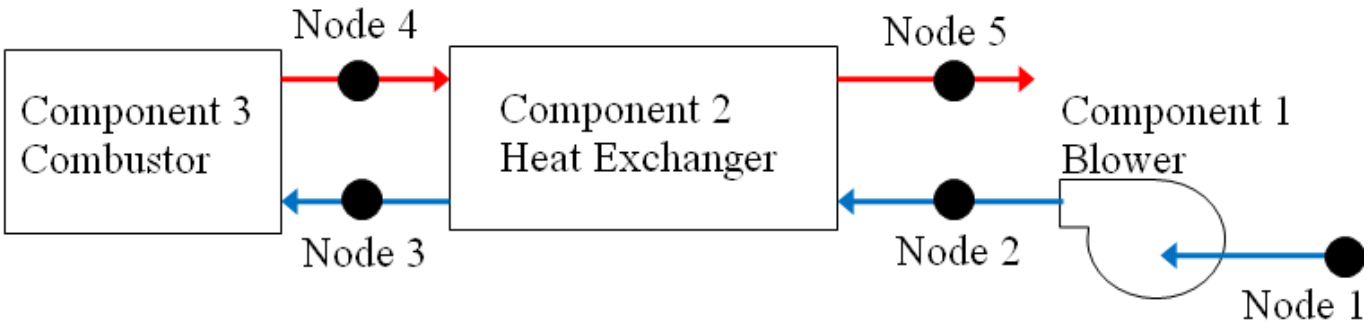

Figure 3.1: Components Connected by Fluid Nodes

While components can be connected to as many nodes as deemed necessary in the TEPSS environment, a single node may only form a bridge between two components or between one component and the environment. Branches and flow path bifurcations can be handled through the use of a splitting component rather than by connecting more than two components to a single node.

The user supplies a host of inputs defining the component parameters, the arrangement of the component/node interconnections, optimization information and cost function definition. All of these are discussed in Section 3.2.

3.2 User Inputs

Since TEPSS is being designed to simulate and optimize any sort of energy system, a whole host of user inputs are required to adequately define a solvable energy system. Component parameter settings, simulation information such as component/node configuration, initial guesses and boundary conditions as well as optimization information including design variables, constraints and convergence criteria must all be specified by the user prior to executing the simulation and optimization routines.

3.2.1 Parameters

Each unique component definition stores a set of internal component parameters specified by the user to fully define the functionality of the component. The component engineering model describes the relationship between node variables attached to the node. These relationships are functions of the node variables themselves as well as the component parameters. It is going to become the goal of the optimization routine to optimize one or more of the component parameters in the system to minimize the objective function.

A structure *parameters* must be provided by the user. Each individual component in the system is assigned a field of the structure. The fields *parameters.component1*, *parameters.component2* etc. are structures themselves, containing the information relevant to defining their respective components. For example, *parameters.component1.field1* stores the value of a parameter that the user intends to pass to component 1 to define the functionality of that component. The design variables that will be manipulated by the optimization algorithm must exist within the structure *parameters*, along with all fixed parameters. Their values will be treated as the initial guess (starting point) when optimization commences.

Carefully note that node variables are not stored within the component but rather on the nodes connecting components. Components have access to data stored on adjacent nodes. Examples of component parameters are geometric dimensions, material properties or environmental effects on the component.

The system in Figure 3.1 has three components: a blower, a heat exchanger and a heater. Each must be assigned a unique name. For example, the user may choose to name these three components ‘*blower1’*, ‘*heatx1’* and ‘*heater1’*. The user could then define the *parameters* structure for the example system as follows:

parameters.blower1.field1 = 20;

parameters.blower1.field2 = 0.75;

… and so on for all blower parameters

Any number of fields may be specified for a component. *field1* and *field2* are intended to represent intrinsic component parameters such as power, geometry, efficiency or any other component parameter that is not a node variable. The field names may reflect their meanings, in which case *parameters* can be defined for all components as:

parameters.blower1.power = 20;

parameters.blower1.efficiency = 0.75;

... and so on for all blower parameters.

parameters.heatx1.component_height = 0.05;

parameters.heatx1.fin_density = 50;

parameters.heatx1.fin_thickness = 0.003;

parameters.heatx1.component_width = 0.25;

... and so on for all heat exchanger parameters.

parameters.heater1.Qin = 10e6;

... and so on for all heater parameters.

Units in the example above are expressed according to the convention established on page v. This convention states that all units are expressed in terms of the elementary SI units of kilograms, meters, seconds, Kelvin, coulombs, radians and their combinations.

3.2.2 Solver Inputs

The user must also define the nodes, component interconnections via nodes, initial guesses, boundary conditions and other simulation parameters. This is done with the *solver_inputs* structure. Table 3.1 contains names, descriptions and examples of all

required fields within this structure. The examples provided are relative to the diagram in Figure 3.1. Each entry in Table 3.1 is a user input required for system concept generation.

Table 3.1: Fields of *solver_inputs*.

Field	Description	Example and More Information (MATLAB notation)
solver_inputs.fstr	String containing text to develop a cell array of components.	*solver_inputs.fstr = '{blower(parameters.blower1) , heat_exchanger(parameters.heatx1), heater(parameters.heater1)}'*
solver_inputs.n	Cell array of nodes. Each node is an object of a class defining the node's domain.	*solver_inputs.n{1} = fluid*(inputs – if needed). Where fluid.m is a class definition file for a domain in the current directory. This expression defines node 1 as a fluid node. Details later in this section.
solver_inputs.cnmap	Links components to nodes. 2-D array containing only integers.	*solver_inputs.cnmap* = [2,1,2,0,0; 4,2,3,4,5; 2, 3,4,0,0]; Details later in this section.
solver_inputs.bcmap	Define boundary conditions of the system.	*solver_inputs.bcmap* = [1,3,101300; 1,2,300; 5,3,101300]; Details later in this section.
solver_inputs.xguess	Initial guesses for node variables that are not fixed by a boundary condition.	*solver_inputs.xguess* = [1,1,1; 2,1,1; 2,2, 310; ... for all initial guesses]; Details later in this section.
solver_inpust h	Small value used to calculate numerical derivatives.	*solver_inputs.h* = 1e-7 Numerical of each node variables with respect to each governing equation will be calculated using the centered difference method and a relative step size of 10^{-7}. If the node variable value approaches zero, the step size is set to *solver_inputs.h.*
solver_inputs.eps	Small number > *solver_inputs.h* that is used to determine if the iteration has converged to a steady state solution.	*solver_inputs.eps* = 1e-6 The recursive Newton's Method algorithm will stop iteration and report the steady state solution when the norm of the previous step size is less than 1×10^{-6}. AND the norm of the residual vector 'e' is less than 1e-6.
solver_inputs.minmax	Enter *'min'* for minimization routine, enter *'max'* for maximization.	*solver_inputs.minmax* = 'min';
solver_inputs.removable	Quickly remove one or more components to establish a baseline performance scenario.	*solver_inputs.removable* = [0,0,0] Details later in this section.

solver_inputs.n

The variable *solver_inputs.n* is supplied by the user as a 1-D cell array containing the sequentially ordered nodes of the system. For a general node, the syntax is *solver_inputs.n*{*i*} = *domain*(*input1*,*input2*…*inputn*) where *domain* is the name of the class definition file for the domain of that node and *i* is the node number. A domain constructor may or may not require inputs, depending on the domain. The domain class definition file for the electrical domain is electrical.m. An example is given here:

EXAMPLE:

solver_inputs.n{1} = *electrical*

creates node 1, a node of the electrical domain.

Domains developed for distribution with TEPSS do not require any inputs, with the exception of the domain class *fluid*. See Section 3.4 for details specific to creating nodes of type *fluid*. Table 3.2 shows the set of domains distributed with TEPSS and associated node variables.

Table 3.2 Domain Names and Variables

Domain	**Variables**
Fluid	mass flow rate, specific enthalpy, absolute pressure
Mechanical (rotational)	torque, angular velocity
DC Electrical	voltage, current

solver_inputs.cnmap

During system setup, the algorithm uses the user-defined component-to-node map (*solver_inputs.cnmap*) to determine which components are connected to which nodes. The component-node map is formulated by the user in the following manner:

1) *solver_inputs.cnmap* is a 2-D array with dimensions *m* by *n+1* where *m* is the number of components in the system and *n* is the maximum number of nodes attached to any one of the components in the system. Each component is numbered according to its order of appearance in *solver_inputs.fstr*. See the example shown later in this section. The first row of *solver_inputs.cnmap* contains information pertinent to the nodal

connections of component #1 and the m^{th} row will contain information for the nodal connections of the last system component.

2) The first column contains an integer specifying the number of nodal connections exist on the corresponding component.

3) Each successive column contains a node number to which that component is connected. If there are more columns available than are needed to list all of the node connections for a component then a zero is used as a placeholder.

4) Each node number should appear in no more than two rows.

EXAMPLE:

solver_inputs.cnmap =

```
[2 1 2 0 0;   %component 1 is connected to two nodes: node 1 and node 2.
 4 2 3 4 5;   %component 2 is connected to four nodes: nodes 2, 3, 4 and 5.
 2 3 4 0 0];  %component 3 is connected to two nodes: node 3 and node 4.
```

Component-node map for 3 components and 5 nodes corresponding with the system configuration pictured in Figure 3.1.

See Section 2.3.3 for an introduction to through and across variable terminology. While most across variables like absolute pressure and temperature are always positive and have no direction at a node, through variables like mass flow and electric current have a direction. The user may wish to track this direction by the sign of the node variable on that node. TEPSS allows users to do this by specifying expected inlets on a component as a positive value of the node number in *solver_inputs.cnmap* and expected outlets as the negative. Note that node variables stored on a node only have a single value. The difference becomes apparent in the component engineering model. In the case of steady mass flow, the sum of mass flow streams entering a component equals the sum of mass flow streams leaving the component. If the mass flows at the expected outlets are taken to be negative, then the sum of all mass flow streams attached to the component equals zero. Take the example system in Figure 3.1. If mass is expected to flow from node 1 to node 5, then *solver_inputs.cnmap* can be formulated as shown.

EXAMPLE:

Let *solver_inputs.cnmap* = [2 1 -2 0 0;
4 2 -3 4 -5;
2 3 -4 0 0];

This information needs to be passed to the components. A convenient way to do so is by appending the rows relevant to each component to their respective fields in the *parameters* structure. For example system in Figure 3.1, this can be done as:

parameters.blower1.direction = solver_inputs.cnmap(1,2:3);
parameters.heatx1.direction = solver_inputs.cnmap(2,2:5);
parameters.heater1.direction = solver_inputs.cnmap(3,2:3);

Using this method to track the sign of through variables will affect the form of the component model equations for through variables. The changes that need to be made are discussed in Section 4.2.

solver_inputs.bcmap* and *solver_inputs.xguess

Each node variable in the system must be either fixed to the value of a boundary condition or assigned an initial value (guess) by the user. Boundary conditions and initial guesses are provided in the arrays *solver_inputs.bcmap* and *solver_inputs.xguess*, respectively. Both arrays contain strictly numeric values. The data is provided in a specific format for both arrays. Each array contains three columns and the number of rows equal to the number of node variables being set in that respective array. The first column contains the node number where the target node variable is stored. The second column contains a number that represents a specific node variable for that respective node. Each node contains a method *update* which assigns a unique reference number to each node variable on that node; the number specified in column 2 must match the number corresponding to the target node variable. See the discussion of the *update* method in Section 4.3 for additional information regarding node variable reference numbers. The third column contains the value to which the specified node variable on the specified node is to be set. An example is shown as it relates to Figure 3.1.

EXAMPLE:

solver_inputs.bcmap = [1, 3, 101300; %set node variable 3 on node 1 equal to 101300.
1, 1, 20; % set node variable 1 on node 1 equal to 20.
5, 3, 101300];% set node variable 3 on node 5 equal to 101300.

On *fluid* nodes, node variable 1 is mass flow, node variable 2 is absolute temperature (or vapor quality for saturated liquids) and node variable 3 is absolute pressure. In relation to Figure 3.1, the set of boundary conditions shown in the example sets the system inlet and outlet pressures equal to 101300 Pa and sets the system inlet mass flow rate equal to 20 kg/s. A similar array *solver_inputs.xguess* must also be defined. The same format is used, but the third column contains initial guess value instead of a fixed boundary condition. Initial guesses must be provided for all node variables not included in *solver_inputs.bcmap*. Node variables that are fixed by *solver_inputs.bcmap* must not be given guess values in *solver_inputs.xguess*. The number of rows in *solver_inputs.bcmap* plus the number of rows in *solver_inputs.xguess* must add to equal the total number of node variables contained in all nodes on the system. Additionally, the number of rows in *solver_inputs.xguess* must equal the total number of equations in all components in the system.

3.2.3 Optimization Inputs

While the *parameters* structure defines internal component properties and the *solver_inputs* structure defines component types and configuration, additional user inputs are required to completely define the behavior of the optimization routine. First, the user defines which fields of *parameters* are design variables that can be adjusted by the optimization routine. Additionally, an update relation must be established by the user to relate the structure-based nomenclature used in *parameters* to the array-based notation used by the optimization routine *fmincon*. Boundaries must be placed on the design space to indicate the highest and lowest values of each design variable between which the user would like to minimize the objective function. And finally, optimization options such as

convergence criteria must be specified. Table 3.3 lists and discusses these inputs and shows examples.

Table 3.3: Optimization Inputs

Field	Description	Example and More Information
dvlist	Cell array (1-D) lists a design variable as a string in each cell.	For 2 DVs: *dvlist* = {'*dv1*' , '*dv2*'};
dvupdate	String that contains update relationships for design variables	*dvupdate* = '*dv1* = *obj*.*dvguess*(1); *dv2* = *obj*.*dvguess*(2)'; Details later in this section.
cost_function_def	1x9 cell array, each slot contains information pertinent to the cost function formulation	Cost function is formulated as: $cost = \frac{\Sigma[A] + t*\Sigma[B] + t*\Sigma([C]*[U])}{\Sigma[D] + t*\Sigma[E] + t*\Sigma([F]*[V])} + \Phi$ See the next subsection for a detailed discussion of the cost function inputs *A-F*, *U*, *V* and *t*.
ub	Upper bounds on DV's	1 x *n* array where *n* is the number of design variables being optimized, slot 1 corresponds to the upper bound on *dvlist*{1}, slot to corresponds to *dvlist*{2} etc. *ub* = [10, 0.01]; conveys that the upper bound on *dvlist*{1} is 10 and the upper bound on *dvlist*{2} is 0.01
lb	Lower bounds on DV's	Same as *ub* but pertaining to the lower bounds of each design variable.
fminconoptions	adjust convergence criteria for optimization routine	See MATLAB documentation for 'fmincon' for a list of convergence criteria and options that can be adjusted and the synax for doing so.
discrete	If the optimization problem contains discrete design variables, specify the discrete values that the variable may have.	*discrete* = { [ ], [1,2, 7.25], [ ] } For a system with 3 design variables, the first and third are continuous and the second can only have the values 1, 2 or 7.25.

dvupdate

The optimization algorithm iteratively selects new points at which to evaluate the cost function. These points correspond to the values of the design and they need to supersede the initial guess provided in *parameters*. An update relation established by the user (*dvupdate*) is used as a bridge between the structural notation used in the *parameters* structure and the array notation used by the *fmincon* routine. It is a single line of string code, but when input into MATLAB's *eva*() function it produces a set of commands. In general, one command per design variable is produced by executing *eval*(*dvupdate*). The command updates the design variables' values using the following syntax (MATLAB notation):

dvupdate = 'parameters.comp1.dv1=obj.dvguess(1) *; parameters.comp2.dv2=obj.dvguess*(2)*;';*

Note the use of semicolons within the string quotations, producing two commands when *eval*(*dvupdate*) is executed. As an example, take the system presented in Figure 3.1 and let the design variables be the heat added by the heater (*parameters.heater1.Qin)* as the first design variable (*dvlist*(1)) and the heat exchanger fin density (*parameters.heatx1.fin_density*) as the second design variable (*dvlist*(2)). The variable *dvguess* is generated by executing *eval*(*dvlist*). To generate an appropriate update relation, the user would enter the following update relation exactly.

EXAMPLE:

dvupdate = 'parameters.heater1.Qin = obj.dvguess(1)*; parameters.heatx1.fin_density = obj.dvguess*(2)*;';*

Each design variable is a property of a component. Since those properties are specified by the user in *parameters* and sorted by component (comp1, comp2 etc.) then the design variables (*Qin* and *fin_density* in the previous example) can be set and reset by executing *eval*(*dvupdate*) at the start of each system simulation. In order to access those properties, the prefix '*obj.*' must be added as shown in the example. After an iteration of the optimization routine, a new set of design variables is suggested by the optimization function in the form of a 1-D array fitting the dimensions of dvguess. These quantities must be delivered to the components, superseding their corresponding previous guess values. Executing *eval(dvupdate)* accomplishes this task, serving as a bridge between the structure nomenclature used in *parameters* and the array notation used by *fmincon*.

3.2.4 Cost Function Formulation

As the cost function is evaluated, the optimization routine's cost function is calculated by summing cost data provided by each component. Some cost data is reported by components in terms of dollars, some is reported in terms of dollars per year, and some is reported in terms of some sort of consumption or production per year (power, fuel, emissions etc.). In order to sum the information, all of the relevant data must first be

expressed in common units. To accomplish this, data reported in dollars per year is multiplied by the system lifetime and consumption per year figures are multiplied by a cost per unit of consumption and the system lifetime. Current TEPSS functionality requires assumptions that the value of money over the system lifetime is constant and the cost of fuel inputs is also constant.

A system that is being analyzed may produce and consume different forms of energy, not all of which are associated with consumption costs. For example, a power cycle may consume chemical energy (fuel) to produce mechanical energy, thermal energy and flow energy. Different forms of energy may have different costs (coal, electricity, etc.), and some forms of energy may have no cost associated with them.

In order to achieve a common unit for cost function evaluation, each energy term is multiplied by its respective cost. The user provides the costs of each form of energy. The prescribed method is to define a cost per unit of consumption in the following form at the user interface: *costperenergy* = [*AC, DC, gas, oil, coal, thermal, flow, KE, PE*]; the standard unit is $/kWh. In each slot the user must provide the cost per unit of that particular form of energy. The same is done for emissions costs: *costperemissions* = [*CO2, NOx, SOx*]; where each slot contains the cost per unit of emissions associated with the respective emission if such a cost exists. The standard unit is kg/s. Users can add other cost figures to the default arrays or prescribe their own consumption cost arrays, but manipulation at the component level will then be needed for each component in the system. See Section 4.2 on component cost functions for more details.

Cost information is reported to the optimization shell by each component in the form of a structure *component_cost* containing four fields *cost, power, emissions* and *physcon* as per the component *cost* method discussed in Section 4.2. The user may wish to select any number of these cost outputs and process them into a suitable cost function for the optimization routine to minimize or maximize. A versatile formulation for the cost function has been devised to allow a wide range of cost metrics to be used in optimization without having to change the component class definitions. The cost function relies on the user's inputs to determine the specific cost metric to be minimized. All cost functions must be derived from the general cost function form shown in eq. (3.1).

$$cost = \frac{\sum_{i=1}^{n}[A_i] + t\sum_{i=1}^{n}[B_i] + t\sum_{i=1}^{n}([C_i]\times[U])}{\sum_{i=1}^{n}[D_i] + t\sum_{i=1}^{n}[E_i] + t\sum_{i=1}^{n}([F_i]\times[V])} + \sum_{i=1}^{n}\Phi \qquad (3.1)$$

n = number of components in the system

Where *'A','B','C','D','E'* and *'F'* are values or arrays in any field of the *component_cost* structure. *U* and *V* are multipliers such as unit conversions, cost per unit fuel, cost per unit emissions etc. They must be defined by the user at the user interface. Also, the variable *t* represents a time multiplier. The value of *t* must be set by the user at the user interface.

The cost function formulation in eq. (3.1) employs a method of pseudo-constrained simulation to place constraints on otherwise unconstrainable aspects of the system. See the discussion of SUMT methods in Section 2.4. The symbol Φ in the cost function formulation represents an exterior penalty function (see eq. (2.7)). Such penalty functions can be written into the component level cost calculation routine *cost*, which is a method that is common to all components in TEPSS. The value of Φ should be set to zero when all pseudo-constraints are satisfied. If a component level value that is not a design variable falls outside of the feasible range (i.e. negative absolute pressure/temperature or unrealistically high pressure/temperature) then the value of Φ can be assigned a positive value as a function of the amount by which the pseudo-constraint is violated. The addition of Φ onto the cost function will steer the optimization algorithm back into the feasible design space. See the discussion of component design in Section 4.2 for more information regarding implementation of penalty functions. Penalty functions are applied to the case studies in Chapters 5 and 6. Examples are available in those chapters as well. An example of one possible implementation is shown in eq. (5.1).

The sigma signs in eq. (3.1) indicate that the specified value is calculated for each component in the system and then all of the values are summed. Care should be taken if possible to prevent the denominator from going to zero in the design space because the gradient based methods used for optimization are only suited for continuous objective functions with continuous derivatives.

A, B, C, D, E, F, U, V, and *t* are all user inputs that must be provided as part of the input *cost_function_def*. *cost_function_def* which is a 9x1 cell array where each cell

contains a value for the aforementioned inputs. The inputs *A* through *F* go in the first six cells respectively, *U* and *V* go in cells 7 and 8 and *t* is input into cell 9. The inputs *A* through *F* refer to component cost outputs that have not yet been calculated, so their expected names are provided in string form so that their values can be acquired using the *eval*() function during the cost calculation phase of the optimization routine. The values of *U*, *V* and *t* are defined at the user interface and therefore they already exist; these values are provided numerically instead of as strings.

Again, refer to Section 4.2 for a summary of what cost outputs are available from the component. Briefly, the three component cost outputs available for use in *A* through *F* are *component_cost.cost*, *component_cost.power* and *component_cost.emissions*. The field *component_cost.cost* is an array of fixed costs, *component_cost.power* is an array of fuel consumption (in watts) sorted by fuel type and *component_cost.emissions* is an array of emissions (in kg/s). eq. (3.2) shows an example of one practical cost function.

EXAMPLE:

$$\text{cost} = \frac{\text{[total fixed cost of all components]+[Energy cost per system lifetime]}}{\text{[net energy generated per system lifetime]}} + \Sigma\Phi \qquad (3.2)$$

Total fixed cost can be calculated by adding the fixed cost of each component directly, so include fixed cost in *A*. *component_cost.power* contains the energy consumption of each component in watts, so to get the energy cost per system lifetime, enter the quantity in *C*, where it will be multiplied by *U – costperenergy* ($/kWh as defined earlier in this subsection) and *t* the system lifetime (30 years). Units are converted from watts to kWh/yr, yielding the energy cost for the system over its lifetime. Finally, to divide by the energy generated, do the same as for *C* but in *E* where it won't be multiplied by cost per kilowatt hour. Then the user input *cost_function_def* will have the values shown in Table 3.4.

Table 3.4 Example for Defining a Cost Function

***cost_function_def* parameter**	**Value**
A	'*component_cost.cost*(*1*)';
B	'0'
C	'*component_cost.power* * 8.766';
D	'0'
E	'*component_cost.power* * 8.766';
F	'0'
U	*costperenergy*
V	*costperenergy*
t	30

where *component_cost.cost*(1) is the fixed cost reported from each component, *component_cost.power* is the energy consumption/production reported by each component and *costperenergy* is the value discussed earlier in this subsection. The user input *cost_function_def* is then used to calculate the value of *cost* in eq. (3.1) above. This is the cost figure that is output by the *objective_f* method. The *fmincon* optimization algorithm then adjusts the design variables and performs iterations of this entire process. Once the convergence criteria of *fmincon* are met, the optimization terminates and reports the final cost value and optimized design variable values to the user.

3.3 Details of Data Flow and Processing

Referring back to the proposed architecture illustrated in Figure 2.2, the system definition created by the user is passed to the optimization shell. There, data relevant to the optimization process can be stored for later use and data relevant to the simulation process or energy system setup can be passed to the simulation shell. The data structures relevant to simulation are stored and the rest of the data structures are used to create the component and node configuration specified by the user. Initial guesses and boundary conditions are used to iteratively solve the system of equations presented by the component engineering models. Once the system operating state is determined, cost information is passed from each component to the optimization shell. The cost function is evaluated and design variables are adjusted. The system is recreated using the new

component configuration and the process repeats until a minimum of the cost function is found to within the tolerance of the optimization algorithm. This entire process takes place through the execution of several functions discussed herein. To clarify, a user is not expected to manipulate these functions; this section is a detailed discussion of how the simulator and optimization routines work. A detailed flow chart of data flow and processing is given in Figure 3.2. The following sections of this chapter walk through the details of each step in the flow chart.

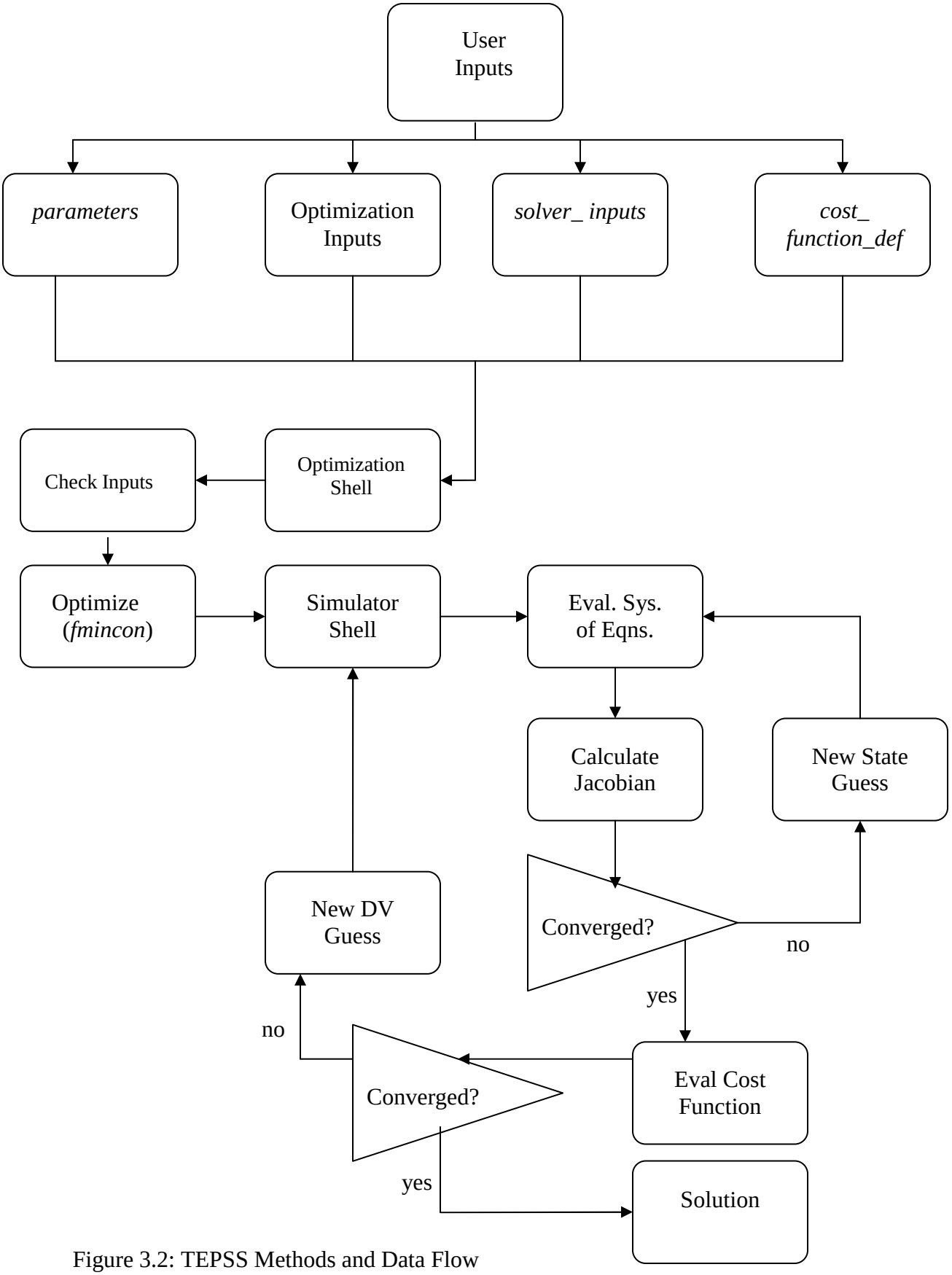

Figure 3.2: TEPSS Methods and Data Flow

3.3.1 System Setup

The system solution and optimization routine is executed by running a script that creates an object of class *optimsolve* and calls the method *optimize* within that object. When this script is executed it collects a series of inputs and creates the optimization shell, an object of class optimsolve, by running the command *C = optimsolve(inputs)* where *C* holds the place of the variable in the MATLAB workspace where the optimization shell is to be stored. Constructing the optimization shell requires five inputs; the inputs are *parameters*, *dvguess*, *solver_inputs*, *dvupdate* and *cost_function_def* – in that order. Each input is defined and discussed in Section 3.2. As the optimization shell is created, the class constructor method stores each input internally as a property so that they can be accessed later by other methods within the object.

Next, the *statecheck* method of the optimization shell is executed to verify that the user has entered the correct number of initial value guesses and boundary conditions. The *numstates* method of each node is called to determine the number of node variables stored in each node. See Section 4.3 for the discussion of domain and node architecture. The total number of node variables is then compared to the number of initial node variable guesses (number of rows in *solver_inputs.xguess*) plus the number of boundary conditions (number of rows in *solver_inputs.bcmap*). If the values are equal then the user has input the correct number of node variable guesses and boundary conditions and the code continues to run. If the numbers do not match, a warning is displayed and the code continues to run, completing the setup phase and executing the optimization algorithm.

3.3.2 System Simulation

When setup of the optimization shell finishes, a call is issued to the *optimize* method of the optimization shell. This routine receives four inputs: *fmincon_options*, *ub*, *lb* and *discrete*. See Section 3.2.3 for information on the formulation and meaning of these inputs. Before the cost function can be evaluated, the system's operating state must first be determined.

The method *objective_f* in the simulation shell serves the purpose of evaluating the cost function. It is called upon recursively by the *optimize* method to simulate the system and evaluate the cost function at the system's changing operating state.

Simulation begins by creating the component objects and storing the appropriate parameters in each component. This is done by executing *eval(solver_inputs.fstr)*. A one dimensional cell array *solver_inputs.f* is produced wherein each cell contains one of the system components. This way the components only receive their own parameters. And as design variables change during optimization, the components can be recreated with the new parameters each time the simulation is run. *objective_f* then creates the simulation shell as a property within the optimization shell object, passing in the structure *solver_inputs*. The simulation shell is an object of class *newtonsolve2* for which the class definition can be found in the file newtonsolve2.m.

The simulation shell constructor performs a host of operations, connecting the components and nodes in the specified configuration and setting the node variable values to the specified boundary condition and initial guess values. Additionally, it stores the *solver_inputs* data structure as a property so that it may be accessed later by other methods within the simulation shell.

The simulation shell constructor applies boundary conditions and initial guesses to nodes by reading from the user inputs *solver_inputs.bcmap* and *solver_inputs.xguess* and then setting the specified node variable equal to the user specified value for each node variable. Node variable values are set by repeatedly calling the *update* method in each node. Once node variable values have all been assigned, *solver_inputs.cnmap* is read and used to create the connections between components via the appropriate nodes. The data stored in a node is available to each component to which that node is attached. If two components are attached to the same node then they can both access the node variables stored within that node, creating the attachment between the components.

Once components and nodes have been created and connected and boundary conditions and initial guesses have been applied, the solver constructor runs a check to verify that the number of node variable guesses supplied by the user in *solver_inputs.xguess* equals the total number of equations in the system. The number of equations in a component is equal to the length of the residual vector that the

component's *compute* method outputs. See Section 4.2 for a discussion of component design, residual vectors and all of the methods contained within components. If the total number of equations contained in all components of the system doesn't match the number of rows in *solver_inputs.xguess* then a warning is displayed stating that there are too many or too few equations in the system engineering model for the number of guesses supplied. If the quantities are equal then the system of equations is square, consisting of *n* equations and *n* unknown node variables. The algorithm for Newton's method requires the approximation of the Jacobian matrix of the system of equations about a guess point and the subsequent calculation of the inverse of that matrix. Without a square system, the Jacobian matrix will not be square and therefore it will be singular. If this check fails the algorithm will ultimately result in an error.

Now that the system has been completely constructed, the solver can begin using the implementation of Newton's Method to isolate a set of node variables that satisfies the system engineering model. In order to do so, the *iterate* method of the simulation shell is executed. The adaptation of Newton's Method takes the initial guess and calculates a step size based on the Jacobian matrix of the system of equations at that initial guess point. The Jacobian matrix is calculated by calling the *jacobian* method of the simulation shell from within the *iterate* method. The inverse of the Jacobian is taken and multiplied by the residual vector to generate a step in the direction of steepest descent toward a root. The residual vector is an array of values by which the engineering model is violated at the current guess. The engineering model is satisfied when all components of the residual vector are zero (within a specified tolerance). Since it comes directly from the component engineering model, this vector is discussed in more detail in Section 4.2, which deals with component design.

The *jacobian* method returns the Jacobian matrix of the system engineering model at the guess location. A Jacobian matrix is a square matrix containing the first partial derivative of each equation in the engineering model with respect to each of the unknown node variables. Partial derivatives are calculated numerically using the centered difference method for each equation in each component with respect to each of the node variables in the system. First the algorithm takes each node variable and adjusts its value slightly toward zero. The relative amount by which the value is changed is specified by

the user in *solver_inputs.h* – a very small value (discussed in 3.2.2). The node variable is divided or multiplied by 1+*solver_inputs.h* to move its value slightly toward or away from zero. The residuals are calculated for each equation for each node variable slightly larger and slightly smaller than the guess value. The centered difference formula (eq. (3.4)) is then used to calculate the partial derivative of that equation with respect to that node variable and the node variable is returned to its original guess value. This data is then processed within the *jacobian* method to formulate the Jacobian of the system. If the value of the node variable approaches zero ($x_j < 10 \cdot$*solver_inputs.eps*), then a fixed step size of *solver_inputs.eps* is used instead of the relative step size of 1+*solver_inputs.h* as shown in eq. (3.5).

Centered difference formula with relative step size:

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(x_j \times (1+h)) - f_i(\frac{x_j}{1+h})}{(x_j \times (1+h)) - (\frac{x_j}{1+h})} \tag{3.4}$$

Centered difference formula with fixed step size:

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(x_j + eps) - f_i(x_j - eps)}{2 \times eps} \tag{3.5}$$

The initial guess vector is taken to be the third column of *solver_inputs.xguess*, which contains the initial values of all of the unknown node variables. With the step size vector calculated, it is added to the previous guess vector and a new guess vector is generated. (See Newton's Method Equations 2.1 and 2.2). The new guess replaces the user supplied guess and the method recursively calculates the new Jacobian, step size and guess vector until the norms of the residual vector and step size vector are smaller than the user defined parameter *solver_inputs.eps*. This set of node variables makes up the steady state operating point of the system to within the number of decimal places of accuracy specified by the user in *solver_inputs.eps*.

A few diagnostic checks attempt to determine whether or not the system of equations is solvable from the qualities of the Jacobian matrix. If the Jacobian matrix is found to have a zero row, it is an indication that the equation corresponding with that row does not depend on any of the system's node variables. A zero column indicates that no

equations in the system depend on the corresponding node variable. If there are no zero rows or columns, but the determinant of the Jacobian is zero, then one or more equations in the system are dependent on others. If any of these checks fail, then the system of equations is unsolvable in TEPSS. Error messages are displayed if these checks determine that the system of equations is unsolvable.

Once the operating state of the system is known, the cost can finally be calculated. Back in the optimization shell the method *objective_f* calls the simulation shell's *cost* method, which retrieves cost information from each component at the current operating point. This information is processed to evaluate the user defined cost function.

3.3.3 Optimization

Calling the *optimize* method of the optimization shell from the user interface executes *fmincon* to minimize a cost function defined using the input *cost_function_def* by adjusting the values of the design variables listed in *dvlist* within the design space specified in *lb* and *ub*. Also, if the user wishes for fmincon to maximize the value instead of minimizing, the user defined value of *solver_inputs.minmax* at the user interface must be changed to '*max*' (default is '*min*'). This negates the cost reported to fmincon by the function *objective_f* so that when *fmincon* minimizes this value, it is effectively maximizing the user-specified cost function.

After the system operating state is determined, cost information is passed to the optimization shell from each component. This data is processed in accordance with the user's cost function input *cost_function_def* to produce a single scalar value representing the feasibility of the system at that operation point. MATLAB's *fmincon* algorithm uses one of three sequential quadratic programming routines discussed later in this section to solve the optimization problem. These routines approximate the second derivative of the cost function. Using this Hessian Matrix, a QP sub-problem is solved to find a search direction and a step size for each design variable is calculated. This process repeats until a minimum is found to within specified tolerances.

In between calls to the simulation shell to solve for the operating point of a new system, the *dvupdate* input is evaluated using MATLAB's *eval()* function, executing the commands specified within the string. These commands replace the previous set of

design variable values in *parameters* with the new set of values determined by *fmincon*. Component objects are recreated at the start of each simulation and the configuration of the new component will take on the values of the new *parameters* structure. Old components are overwritten by their new counterparts with updated parameters.

fmincon is a constrained nonlinear optimization routine developed in MATLAB by The Mathworks Corporation that converts the constrained nonlinear problem into a linearized unconstrained problem using the Method of Lagrange Multipliers (MLM) discussed earlier in Section 2.4. The algorithm used the sequential quadratic programming approach discussed in Section 2.4. MATLAB's *fmincon* optimization routine can use one of three algorithms to solve the optimization sub-problem. The user can specify which one to use by manipulating the user input *fmincon_options* in accordance with the proper syntax. See MATLAB's documentation for *fmincon* for a list of all options that can be set. The algorithms are named *active set*, *trust region reflective* and *interior point*. The default in TEPSS is the *active set* algorithm. Each uses SQP to isolate a minimum of the Lagrangian objective function developed from the user defined cost function and constraints. Such approaches require the use of an approximated Hessian matrix to calculate the step size taken in each design variable dimension. The three algorithms available within *fmincon* mainly differ in how the calculation and recalculation of the Hessian is handled.

The *trust region reflective* method requires the user to supply the gradient of the objective function; it also does not support the use of inequality constraints that are not side constraints. Since a general cost function is used in TEPSS and the user may choose to develop the cost function in an innumerable number of ways, this algorithm is not currently supported by TEPSS. Until such time as a general gradient calculation is included in the TEPSS architecture, the default algorithm for optimization in TEPSS is the *active set* algorithm. This is a medium scale optimization algorithm that uses dense linear algebra and full matrices instead of the potentially faster sparse methods. Using this method to solve large systems can become computationally intensive. In the *active set* method, gradients of the objective function are calculated using the forward stepping finite difference method. This may be adjusted by the user in *fmincon_options* to use a

centered difference method, doubling the time taken in each iteration, but increasing the accuracy of the gradient estimate (see MATLAB documentation for *fmincon*).

3.4 Additional Details

Quasi-Newton Method

In order to satisfy some of the additional requirements for TEPSS, some adjustments to the traditional methods of nonlinear equation solving and optimization had to be made. As previously motioned, the equation solver uses an adapted form of Newton's Method that leverages numerical derivatives in the Jacobian matrix instead of directly calculated analytical derivatives. This is often referred to as a quasi-Newton Method. It is a multidimensional extension of the secant method of solving equations using numerically calculated derivatives in a single dimension.

Also in the solver, several checks have been implemented with associated warning messages to guide users into formulating well posed systems. First, the number of guesses provided for unknown values of node variables is compared to the number of equations supplied in the system engineering model. If the values are equal, then the procedure continues, comparing the total number of node variables in all nodes of the system to the total number of guesses and boundary conditions specified by the user. Again, if the values match, then the procedure continues to try to solve the system of equations for the values of the unknown node variables. In either case, if the values do not match, a warning is displayed and the routine continues, often leading to an error or a violated boundary condition.

Closed Loop Simulations

Additionally, when there is a closed loop configuration of components like the one tested in Case Study I (Chapter 5, Figure 5.1), equation dependence can result. Take the following example of mass flow through three components connected to three nodes forming a closed loop. According to steady state conservation of mass, *mass_flow1 = mass_flow2*, *mass_flow2 = mass_flow3* and closing the loop, *mass_flow3 = mass_flow1*. The third equation can be deduced from the transitive property of the first two equations. As a result, the third equation is dependent upon the first two. This has serious

implications for the equation solving algorithm in TEPSS. The Jacobian matrix will be singular if it is non-square or if it contains a dependent row. The inverse of the Jacobian is needed to find the direction of steepest descent toward a root, therefore having this extra equation in the system of equations will cause the method to fail.

Also, in closed loop systems, there may or may not be enough free node variables to implement necessary boundary conditions. To circumvent the aforementioned failure mode and free up an additional node variable for a boundary condition, a reference component is used. Reference components that have been developed connect two nodes and close a loop of nodes. This component's engineering model contains an additional equation relating an across variable (pressure, voltage etc.) to a prescribed parameter value and it contains no equation relating the relationship of a through variable (mass flow, electric current etc.) entering and leaving the component. A pressure reference component is used in the case study in Chapter 5 (see Figure 5.1) since the pressure in the closed loop in the system does not interact with the environment. See the component models in Section 5.1 for an example of how the reference component's engineering model differs from that of other components.

Dealing with Discrete Design Variables

Outside of the MATLAB algorithm, the user is able to make certain modifications to the optimization scheme to help ensure that the solution is correct. The *fmincon* algorithm is designed for use only with continuous design variables. If one or more design variables can only hold discrete values, the user may specify which variables and which possible values by listing the feasible values for each design variable within the user defined parameter *discrete* (discussed in 3.2.3). Listing no values implies that the design variable is continuous throughout the specified design space. In cases where one or more design variables are discrete, the *fminconset* algorithm is used to try to find the minimum of the objective function satisfying the discrete requirements. This is done by first solving the optimization problem as though all design variables are continuous, then searching in the vicinity of that minimum for the minimum that satisfies the discrete requirements of the subset of design variables that are not continuous. The *fminconset* function is available for download free of charge from the MATLAB website [27].

SUMT Methods for Constraining Values Other than Design Variables

The user may also wish to limit the physical magnitude of the simulation solution due to engineering/material strength constraints. If the parameter is a design variable, it may be constrained by side constraints on the design space. However this can not be done for node variables or variables calculated as functions of other component parameters. In this case, a penalty function can be added to the objective function forming a pseudo-objective function. This becomes the new cost function to be minimized. For example, the user may wish to limit the electrical current exiting a voltage source; this can be accomplished by specifying the maximum current as a parameter of the voltage source component. That current then is passed into the component cost function, where it is compared with the solution exit current. If the solution current is greater than the user specified maximum current, a penalty, proportional to the square of the difference between the solution current and maximum current, is added to the original cost function. The SUMT methods discussed in Section 2.4 are used to accomplish this. Examples are available in Section 4.2 and an application is shown in Equation 5.1.

Avoiding Simulation Failure due to Physical Impossibilities

While each individual design variable can be constrained with upper and lower bounds to define a particular design space, some points within the design space may not have simulation solutions if some function of the design variables used in the simulation returns an impossible value. For example, two lengths may be positive but the resulting cross sectional area formed by the lengths may be too small to allow air to flow through that area at a fixed mass flow rate with the flow energy available. In this case the simulation routine will not converge and the solution cannot be used to evaluate the cost function, so the pseudo-objective approach described earlier will not work.

An alternative method is employed to detect these faults ahead of time and persuade the optimization algorithm to return to feasible space. If a component has failure modes like the one described here, the user is encouraged to program a check into the component model, so that the simulation does not try to run the simulation routine with that set of design variables. If the check returns that the simulation is unsolvable given

the current set of design variables, then the simulation is skipped and the cost function is set to a large fixed value during minimization (large negative value during maximization). The magnitude of the large fixed value is determined on a system to system basis depending on the order of magnitude of the cost function in previous successful simulations during the same optimization run. Currently the large value is set to be several orders of magnitude larger than the first value of the cost function. If no initial value for the cost function is available (1^{st} iteration), then the routine stops and an error message is displayed.

Special Considerations for *fluid* Domain

Due to the prevalence of fluid systems in energy cycles, some special considerations have been made to accommodate the thermohydraulic node domain known as *fluid* in TEPSS. The file fluid.m contains the class definition for *fluid* nodes. The thermodynamic state of a fluid can be determined knowing the values of two independent thermophysical properties. Instead of directly incorporating curve fits for numerous properties for numerous fluids into TEPSS, the third party program FluidProp [28] is leveraged to calculate a host of thermodynamic properties given the values of two independent ones. FluidProp is an activex server that runs in the background of TEPSS; objects of class *FluidProp* are created automatically by the constructor of fluid nodes and called upon to calculate thermodynamic properties given the fluid state. See reference [28] for documentation detailing the functionality of this third party program.

The *fluid* domain is different from the others developed for TEPSS, because it can be used to describe the behavior of a host of different types of fluids, multiphase fluid flow applications and mixtures of different compounds. FluidProp is capable of handling these complexities, but additional inputs are required from the user when creating a node of type ‘fluid’. Nodes of type fluid are defined with four inputs in the following manner:

$$solver\_inputs.n\{i\} = fluid(comp, ratio, database, states); \tag{3.6}$$

where *i* is the node number. The input *comp* is a string containing the names of the fluids contained on that node using the FluidProp syntax for the names of the fluids.

EXAMPLE:

comp = 'NH3,CO2'

denotes that the node contains a mixture of ammonia and carbon dioxide. *NH3* and *CO2* are FluidProp short names for these fluids. A complete list of fluids available in each FluidProp database is available in the FluidProp documentation. Fluids specified here must exactly match the FluidProp name for the fluid [28]. Take care not to type an extraneous space after a comma.

The input *ratios* is a one dimensional array specifying the relative ratios of each component in the fluid mixture. The first entry corresponds to the first fluid name appearing in *comp* and so forth. The length of *ratios* should equal the number of components specified in *comp* and *sum(ratios)* should equal 1. An example is shown for a 60/40 mixture of ammonia to carbon dioxide.

EXAMPLE:

ratios = [0.6, 0.4];

assuming *comp* = 'NH3, CO2'

The input *database* is a string corresponding to one of the five available FluidProp property databases. The only possible (case-sensitive) inputs are '*TPSI*', '*StanMix*', '*GasMix*', '*IF97*' (water only) and '*RefProp*' (purchased separately, currently unsupported by TEPSS). Each database has specific advantages and disadvantages, including the fluids for which properties are available in each database, the available thermodynamic properties, and the pairs of independent properties that can be used to define the state of the fluid. These databases were not developed by the developers of FluidProp, but rather they are accessed by the FluidProp activex server to retrieve the requested properties. More information is available in the FluidProp documentation [28].

The final input *states* is a string that tells TEPSS which two independent states are being specified by the user to look up the initial thermodynamic state of the node. Possible inputs are '*PT*' (pressure and temperature) and '*Pq*' (pressure and vapor quality). All string inputs in this section are case sensitive. Specific enthalpy is not used to define the initial thermodynamic state because the reference state used by FluidProp to calculate the specific enthalpy varies from one fluid to another.

In the fluid domain distributed with TEPSS, mass flow, specific enthalpy and pressure are the node variables at each fluid node. Pressure and specific enthalpy are independent thermodynamic properties, which can be used to calculate other important properties like temperature, specific heat, thermal conductivity, and density. FluidProp has access to a set of databases containing these properties for a host of fluids and mixtures of fluids as a function of the thermodynamic state of the fluid. See Section 4.3 for more information on calling FluidProp to retrieve the thermodynamic properties of a fluid at a given thermodynamic state.

Unfortunately, since specific enthalpy is measured relative to a reference state, rather than on an absolute scale, using enthalpy to define boundary conditions and initial guesses is less than ideal, because the reference state used by FluidProp will often differ from the reference state used by the user. After a brief investigation, it was discovered that the specific enthalpy reference states in FluidProp vary from one fluid to another and the reference state is not sufficiently easy to determine. To deal with this, fluid domain node variable guesses and boundary conditions are provided in terms of two absolute independent state variables, one of which is pressure and the other is either absolute temperature or vapor quality depending on an intuitive analysis of the system. Often, the thermodynamic state of a fluid can simply be defined by temperature and pressure. The exception of course is with multiphase fluids, where temperature and pressure are not independent thermodynamic properties and therefore do not specify a unique thermodynamic state. However, in multiphase fluids vapor quality is an independent design variable and can be used as the second property along with pressure to define the state of the fluid.

The following rules have been adapted for defining fluid node variable guesses and boundary conditions in the user inputs *solver_inputs.xguess* and *solver_inputs.bcmap*. If a guess is being supplied, define the *fluid* node input *states* as '*PT*' (see Section 4.3 discussion of node design), supply the absolute pressure guess as usual and a guess for absolute temperature in place of a specific enthalpy guess (examples below). The temperature and pressure supplied will be used by FluidProp to look up an initial guess for the specific enthalpy of the fluid at that state. Do the same for boundary conditions, where flow is expected to be single phase. For multiphase boundary

conditions, define the *fluid* node input *states* as *'Pq'* and supply the vapor quality in place of the specific enthalpy as a boundary condition. At the conclusion of a simulation, the node variables are all known, including the specific enthalpies at fluid nodes. These values are converted back to either temperatures or vapor qualities using FluidProp and reported to the user, eliminating the need for the user to supply or interpret specific enthalpy values that correspond to the same reference state as the FluidProp software for that fluid. This feature allows TEPSS to handle phase changes and multiphase flows without requiring the user to match his or her reference state for specific enthalpy to the one used by FluidProp.

In a component that is attached to fluid nodes, thermophysical properties of the fluid on the node can be determined by calling the *getprop* method of the *fluid* domain. This method is unique to the *fluid* domain. If a property such as density or specific heat is desired for calculation, the user can leverage this method with the appropriate inputs to retrieve that value. *getprop* is called as follows:

value = noden.getprop(property_name, states, state1, state2)

where *noden* is the name of the node on which the user is interested in obtaining one of the thermophysical properties of the fluid, *property_name* is the FluidProp name of the property the user wishes to calculate, *states* is the two character string array specifying which two thermodynamic states will be used to determine the third one (*value). state1* is the value of the first of the two states specified in *states* and *state2* is the second. See FluidProp documentation [28] for a more detailed explanation.

Chapter 4

Guide to Expanding TEPSS

TEPSS is intended to be a widely accessible and expandable tool for the simulation and optimization of energy systems. While it is not currently distributed as a standalone software tool, it leverages the widely used MATLAB software including the MATLAB Optimization Toolbox. In addition, the FluidProp add-on for calculating the thermodynamic and transport properties of a fluid is available for free [25]. If mixed integer problems are going to be solved, the free fminconset.m [27] script must be downloaded and put in the MATLAB directory containing the execution file. TEPSS is able to achieve its full functionality with these software packages and an appropriate set of user inputs and component models. The major focus of this chapter is to guide the user in designing new components and domains that will be compatible with the TEPSS environment.

4.1 Operating TEPSS Software

Minimum requirements for TEPSS are MATLAB 2008a/b or later with the MATLAB Optimization Toolbox add-on and FluidProp for Windows XP 32-bit or later. FluidProp currently does not work with the 64-bit version of MATLAB, as a result, TEPSS is currently limited to use with 32 bit versions of the program. And finally, an appropriate amount of computing power is required depending on the size of the system being simulated. Systems that contain components with finite element models have been found to slow the simulation process considerably. In Case Study I (Chapter 5) a combined cycle system with 27 equation and 27 unknowns is optimized in two dimensions on a machine with a dual core Intel® processor with 3 GB of RAM in a matter of a couple of minutes. Case Study II (Chapter 6) took considerably longer due to the finite element model contained in the thermoelectric power unit component. Optimization of the two selected design variables took several hours rather than a couple of minutes.

TEPSS is executed from a script (.m) file in MATLAB. The inputs described in Section 3.2 must be provided by the user in the appropriate manner and the system component model must be solvable for all unknown node variables given the set of boundary conditions provided by the user. Once these criteria are met, TEPSS has been shown to perform according to specifications (Chapters 5 and 6). The appropriate format for supplying user inputs is described in detail in Section 3.2, with special considerations for defining fluid nodes in Section 3.4.

4.2 Designing Components for TEPSS

TEPSS will be distributed with a base package of components, node domains and a template script for execution. Most of the component engineering models in the base package are largely based upon first principles of thermodynamics, providing adequate complexity for validation of the software, but lacking the complexity needed for industrial applications requiring detailed analysis. TEPSS was designed from the start to be a flexible and expandable tool intended for use with any steady state engineering model. If TEPSS is put to use in industry, it is expected that the user will develop and implement additional component models as needed.

Component models are objects whose properties and methods are defined in class definition files. All component class definitions inherit the parent class *handle*. This allows multiple copies of the same component to be present in an energy system at a time. The file name must match the class name.

Components all contain four core methods, common to each component. They are: constructor, *compute*, *cost* and *paramcheck*. The constructor serves the single purpose of storing the parameters of the component as a property when the component is created. The name of the constructor method must exactly match the name of the component or it will not be called automatically when the component is created. The *compute* method contains the engineering model of the component. The *cost* method reports the cost parameters associated with the component to the optimization shell once the values of the node variables have been determined. Finally, the method *paramcheck* contains a user defined check intended to determine whether or not the component may exist as it is defined. This is done to prevent attempts to simulate impossible systems.

Such systems may result from negative area or length dimensions or a host of other phenomena that make a system impossible. The methods *compute*, *cost* and *paramcheck* must be named as such. An example component class definition code is provided in Figure 4.1 for a simple heat exchanger component. Since the class definition is *heatx*, the file must be named heatx.m.

```
classdef heatx<handle
    properties
        parameters
        onoff
    end
    methods
        function obj = heatx(parameters)
        obj.parameters=parameters;%store parameters for use by other methods
        end
        function e = compute(obj, node1, node2, node3, node4, onoff) %compute method

            %name the nodes so that equations are easy to read
            hfluidin=node1; %hot side inlet node
            hfluidout=node2;%hot outlet node
            cfluidin=node3; %cold inlet node
            cfluidout=node4;%cold outlet node
            obj.onoff = onoff;
            %Calculate log mean temperature difference
            if strcmp(obj.parameters.flowdir, 'parallel')==1 %for a parallel flow setup
                            dtln=(((hfluidin.temp - cfluidin.temp) -...
                                (hfluidout.temp - cfluidout.temp))...
                                /log(abs(hfluidin.temp-cfluidin.temp)/...
                                     abs(hfluidout.temp-cfluidout.temp)));
            elseif strcmp(obj.parameters.flowdir, 'counter')==1%for a counter flow setup
                            dtln=(((hfluidin.temp - cfluidout.temp) -...
                                (hfluidout.temp - cfluidin.temp))...
                                /log(abs(hfluidin.temp-cfluidout.temp)/...
                                     abs(hfluidout.temp-cfluidin.temp)));
            else
            end

            Qhx = obj.parameters.UA*dtln;

            e(1) = cfluidin.mdot*sign(obj.parameters.direction(1))+...
                cfluidout.mdot*sign(obj.parameters.direction(2)); %mass is conserved
            e(2) = cfluidin.mdot*sign(obj.parameters.direction(1))*...
                (cfluidout.enthalpy-cfluidin.enthalpy)-Qhx; %energy is conserved
            e(3) = cfluidin.press - cfluidout.press;  %assume no pressure drop
            e(4) = hfluidin.mdot*sign(obj.parameters.direction(3))+...
                hfluidout.mdot*sign(obj.parameters.direction(4)); %mass is conserved
            e(5) = hfluidin.mdot*sign(obj.parameters.direction(3))*...
                (hfluidin.enthalpy-hfluidout.enthalpy)-Qhx; %energy is conserved
            e(6) = hfluidin.press - hfluidout.press; %assume no pressure drop
            obj.parameters.pressinc = cfluidin.press;%store for use in cost
        end
```

Figure 4.1a: Properties Sample Code for Component Class Definition

```
46
47              function component_cost = cost(obj)
48                 component_cost.cost = [45+9*obj.parameters.UA,0]; %fixed cost
49                 component_cost.power = [0,0,0,0,0,0,0,0,0]'; %no power produced/consumed
50                 component_cost.emissions = [0;0;0];
51                  if obj.parameters.pressinc > obj.parameters.pressmax
52                      component_cost.physcon = 0.01*...
53                          (obj.parameters.pressinc - obj.parameters.pressmax)^2;
54                            %penalize the cost function if the inlet pressure exceeds a
55                            %prescribed maximum value
56                  else
57                  component_cost.physcon = 0;
58                  end
59              end
60
61              function y = paramcheck(obj)
62                  if obj.parameters.UA>0 %if UA is positive proceed with simulation
63                      y=0;
64                  else
65                      y=1; %otherwise skip simulation
66                  end
67              end
68          end
69      end
```

Figure 4.1b: Sample Code for Component Class Definition

Component Method: *compute*

The engineering model of a component is a set of equations linking the node variables of the attached nodes to one another through some sort of mathematical relationship based on the physical behavior of the component. This model is defined in every component within the *compute* method. This method is common to all components and when called, it receives the adjacent nodes as inputs and it outputs a one dimensional residual vector containing a single scalar value for each equation. Residual equations are the core of the *compute* method. They can be derived from physical laws (i.e. conservation of mass), empirical relationships or finite element analyses. A residual equation is formulated as follows:

$$e(i) = f(\bar{x}) \qquad (4.1)$$

where $e(i)$ is a free variable and $f(\bar{x})$ is a function of the node variables, component parameters and any other available information. The physical relationship, empirical model or finite element model is satisfied when $e(i) = 0$. In the above sample code there are six residual equations with the free variables *e(1)* through *e(6)*. The equations on lines 34 and 39 of Figure 4.1 are steady state conservation of mass equations applied to the hot and cold sides of the heat exchanger, the equations on lines 36 and 41 of Figure 4.1 are

conservation of energy equations derived assuming no work is done by the adiabatic heat exchanger.

$$Q_h = \dot{m}\Delta h_{hot} = UA\Delta T_{lm} \tag{4.2}$$

$$Q_c = \dot{m}\Delta h_{cold} = UA\Delta T_{lm} \tag{4.3}$$

where Q_h and Q_c are the respective rates at which heat leaves or enters the hot and cold sides of the heat exchanger, *h* is specific enthalpy and *UA* is the overall heat transfer coefficient of the heat exchanger in units [W/K]. The rightmost equality in each equation is arranged to equal zero and then the free variable *e* is added to produce the equations seen in the code. The equation for log mean temperature difference (ΔT_{lm}) is stated later as eq. (5.3) and calculated in the sample code in Figure 4.1 in lines 19 through 28. The equations on lines 38 and 43 of Figure 4.1 enforce the assumption that there is no pressure drop across the component.

The residual vector *e* consists of a 1-D array containing all residuals, one from each equation in the engineering model of the component. The simulation routine calls the *compute* method for each component and receives back the residuals for all equations in the system's engineering model. The goal of the simulation routine is to reduce all residuals to zero within a user defined tolerance, while in the process determining the node variables that satisfy all of the relationships set forth in the system engineering model.

As discussed in Section 3.2 under the description of *solver_inputs.cnmap*, through variables have a direction, but they are stored as a scalar on the node. To account for this direction, minus signs are added to expected outlets in *solver_inputs.cnmap* by assigning a value to *parameters.direction.* In Figure 4.1, *hfluidout.massflow* and *cfluidout.massflow* can be negated by multiplying by the sign of the appropriate element of *heatx.parameters.direction. heatx.parameters.direction* is the row of *solver_inputs.*cnmap that corresponds to the component *heatx*. This allows the steady state conservation of mass equations associated with *e*(1) and *e*(4) on lines 34 and 39 of Figure 4.1 to be written as they are. Note the use of MATLAB's *sign*( ) function.

Component Method: *cost*

The method *cost* is called by the optimization routine to deliver a common set of outputs to the optimization shell, where the outputs from each component method *cost* are processed to determine the scalar value of the objective function being optimized. The component's cost method outputs the structure *component_cost* which has four fields: *cost*, *power*, *emissions* and *physcon* (refer to the sample code in Figure 4.1 lines 47-59). The first field, *cost*, is an array containing costs that are determined as functions of the component parameters. Each component in the system must output *component_cost.cost* arrays of equal size. The default size is 1x2. In Figure 4.1, the fixed cost of the component is a function of its overall heat transfer coefficient *UA* which is a user specified parameter and potential design variable (line 48).

The field *power* contains an array of terms that state the scalar rate at which the component consumes a particular form of energy. Since the cost of electricity differs from the cost of fuel and other forms of energy, each entry can have a different cost associated with it. The default configuration is a 1x9 array where each entry corresponds to the rate at which energy is consumed in the form specified in Table 4.1. Since the adiabatic heat exchanger in Figure 4.1 neither produces nor consumes any net energy, all fields are set to zero.

Table 4.1: Default *component_cost.power* Indices

Index	Energy Type
1	AC electricity
2	DC electricity
3	Chemical – Natural Gas
4	Chemical - Petroleum
5	Chemical - Coal
6	Heat
7	Pressure
8	Kinetic
9	Potential

In *component_cost.power*, positive entries indicate that the component consumes energy of the type corresponding to the index of the entry in Table 4.1. Negative entries indicate that the component produces energy of that particular form. The intention of including these terms is to account for fuel cost and revenue from energy production. That being said, multiplying the number of units of energy produced or consumed by the cost per unit energy will yield the cost of the energy required to operate each component. Energy production/consumption can be developed from power by multiplying by the system lifetime. This could be a useful piece of data in many cost functions, including the cost function used in Case Study I (eqs 5.1 & 5.2). Setting *component_cost.power* equal to *C* or *F* in the cost function definition is one way to calculate this cost. See Section 3.2.4 for details. The indices in Table 4.1 may be changed or the array may be expanded at any time so long as the corresponding user defined pricing vector (*U* or *V* in the cost function definition) has the same length as *component_cost.power* and all components output a *component_cost.power* of the same length.

The output *component_costs.emissions* serves a similar purpose to *component_costs.power*, but it is intended to be used to calculate the cost of the emissions produced by the component, if any. Cost per unit of emissions is provided by the user. Table 4.2 shows the default indexes of the different emissions outputs.

Table 4.2: Default *component_cost.emissions* Indices

Index	Pollutant
1	Carbon Dioxide
2	Nitrous Oxides
3	Sulfurous Oxides

Again, the array can be expanded or changed at any time, as long as any corresponding user-defined cost matrix is adjusted accordingly. The heat exchanger in the example does not produce emissions, so the fields are set to zero in the example.

The cost output *component_costs.physcon* is used for the sole purpose of penalizing the objective function in the event that a node variable or other non-design variable lies outside of some user-defined feasible range. In the example code above a

penalty function becomes active when the cold side inlet pressure exceeds a value *parameters.maxpress* specified by the user. This method can be useful in levying constraints on non-design variables, especially node variables. Node variables and other calculated values cannot be directly constrained in TEPSS the way design variables can be constrained with side constraints (*lb* and *ub*) in *fmincon*. As a result, TEPSS employs a sequentially unconstrained minimization technique (SUMT) known as an exterior penalty function to persuade the optimization routine to search for minima in the feasible range. Section 2.4 discusses SUMT methods in greater detail. The *cost* method in the component should contain a conditional statement for which *component_cost.physcon* is equal to zero for all cases in which the node variables are feasible solutions to the system of equations and otherwise the value of *component_cost.physcon* is proportional to the square (or some exponent >1) of the amount by which the implicit constraint is violated. In components where these statements are used, it might be required to store the required node variables and calculated values used in the *compute* method as properties of the component object so that the *cost* method can access those values. Figure 4.1 lines 51 – 58 show the implementation of a penalty function if the cold side pressure exceeds a prescribed maximum *pressinc*. Another example is shown in eqs (5.1 & 5.2) in the form of the cost function formulation for Case Study I.

Component Method: *paramcheck*

The final method common to all component class definitions is *paramcheck*. This function is used to try to catch impossible systems proposed by either the user or the optimization routine before the simulation routine begins to run for that configuration. Trying to simulate one of these systems will result in no solution or a solution with values that do not make sense (i.e. negative absolute pressure). The component can conduct checks to confirm physical requirements such as length > 0 and area > 0 into this method. If the user specifies a system for which one or more of these checks are violated initially, the routine stops immediately and displays an error message to the user. However, during optimization, if the sequential quadratic programming (SQP) algorithm tries to simulate a system where one of these checks is not satisfied, TEPSS will try to steer the minimization routine back into feasible space. In this case, simulation is skipped and the

cost function is penalized by a factor proportional to the value of the cost function the first time it was evaluated (design variables = initial guess). The constant of proportionality is large enough that the resulting cost is orders of magnitude greater than at the starting point. This jump in the objective function value forces the optimization algorithm back into the feasible space. Since the location of where the system becomes infeasible is not known in this case, the penalty function is set equal to a constant rather than making it a function of the magnitude of the constraint violation. This causes the first derivative of the objective function in the infeasible space to go to zero. A warning message is displayed when this penalty function kicks in because the loss of derivative information may cause the optimization algorithm to sense false convergence to a minimum.

To set a check in the method *statecheck,* set the output equal to vector of Boolean test values. Let a test output 0 if the system passes a check and 1 if the system fails a check. Then the optimization routine knows that a system is feasible if all components' *statecheck* methods output only arrays of zeros. If there are no tests for a component, set the output of *statecheck* equal to 0. In Figure 4.1, the simulation proceeds normally as long as the overall heat transfer of the heat exchanger coefficient (*UA*) is positive (lines 61 through 67).

Other methods may be incorporated into components, as long as their names do not conflict with the required method names discussed above. It is conventional within TEPSS to only allow components to access information on adjacent nodes and to disallow components access to the parameters of other components to protect data, prevent corruption and allow for future expansion without the requirement to recode existing components. This modularity is maintained as part of the physical system analog upon which the TEPSS platform is predicated and proper performance of the platform is not guaranteed should a user choose to implement methods that violate this condition.

4.3 Domain and Node Design

TEPSS will be distributed with a basic package of domains to which nodes may belong. The important function of a domain is to define and enumerate the node variables that exist on a node of that domain. Simple electrical and mechanical rotational domains

(electrical.m and mechrot.m) have been developed for distribution with TEPSS, along with a more comprehensive thermohydraulic domain fluid.m. If the user should wish to expand upon these domains or introduce additional domains, all of the necessary information required to do so is contained in this section.

Domains are class definition files, and as a result, they have properties and methods. Like components, they inherit the parent class *handle* for the same reason, so that many nodes of the same domain may exist within a system at once. For each domain, there is a distinct set of properties and a set of methods common to all domains, as well as methods unique to that particular domain (as required). The names of the node variables exist as properties of a domain as well as any additional properties that the user wishes to include on nodes of that domain. Methods common to all domains are *update* and *numstates*. An example of a mechanical rotational domain (mechrot.m) is given in Figure 4.2 to aid with the discussion of these methods.

```
1   classdef mechrot < handle %mechanical rotational domain
2       properties
3           torque    %torque is a property
4           angvel    %angular velocity is a property
5       end
6       methods
7           function update(obj, x) % in setup.m bcmap and xguess colums 3
8                                   %require the user to supply a property #
9                                   %to each guess. That guess is interpreted
10                                  %here and assigned to the appropriate
11                                  %state.
12                      if     x(2) == 1
13                          obj.torque = x(3); %torque is property # 1
14                      elseif x(2) == 2
15                          obj.angvel = x(3); %angular velocity is property # 2
16                      else
17                      end
18          end
19          function num = numstates(obj)
20              num = 2; %there are two node variables in this domain.
21          end
22      end
23  end
```

Figure 4.2: Sample Code for Domain Class Definition

Domain Methods: *update*

The *update* method (Figure 4.2 lines 7-18) carries out the function of receiving initial and subsequent guess values and boundary conditions for node variables of nodes

of the domain. *update* is responsible for interpreting those guess values and boundary conditions and applying the appropriate value to the appropriate state on the node. A unique reference number, typically an integer indexing from one, is assigned to each of the node variables that exist in the domain. The *update* method receives two pieces of data, the node variable number and a value to which the node variable is to be set. In the example code, the update method interprets the value in *x*(*3*) as a torque if *x*(*2*) is equal to 1 and as an angular velocity if *x*(*2*) is equal to 2. The input *x* is a row of the array *solver_inputs.bcmap* or *solver_inputs.xguess*. During simulation, this method is called repeatedly to update the nodes with new guesses calculated during each iteration of the simulation process.

Three domains are developed for distribution with TEPSS, *fluid, mechrot* and *electrical*. Table 4.3 gives the node variables and their respective reference numbers for each of these domains.

Table 4.3 Domains, Node Variables and Reference Numbers

Domain	Node Variable	Reference Number
fluid	mass flow	1
	specific enthalpy	2
	pressure	3
mechanical rotational	torque	1
	angular velocity	2
electrical	voltage	1
	current	2

Domain Methods: *numstates*

The method *numstates* receives no inputs and outputs a single integer equal to the number of node variables that exist within that domain. In Figure 4.2 lines 19 through 21, this output is 2 because there are two node variables on each node of the mechanical rotational domain (torque and angular velocity). This simple method is called at the start

of a simulation for each node so that the number of node variables in the system can be compared to the number of guesses and boundary conditions supplied. This check is done to ensure that the user has supplied the correct number of guesses and boundary conditions.

Having established the ability to simulate and optimize energy systems and add to the library of components and domains, the software platform TEPSS is complete. The remaining chapters deal with validation of the simulation and optimization routines within TEPSS. The first case study is a simulation of a published case study of a combined cycle and the second case study involves optimization of a thermoelectric heat recovery platform integrated into a regenerative Brayton cycle.

Chapter 5

Case Study I: Simulation and Optimization of a Combined Cycle

To validate the TEPSS simulation and optimization platforms, a combined cycle based on first thermodynamic principles from a paper by Wicks is chosen for implementation in TEPSS [29]. The combined cycle is chosen because it tests the ability of TEPSS to simulate open and closed circuit systems, fluid phase changes, nodes of varying domains, multiple mass flow streams and components with connections to various numbers of nodes. Components are developed that contain engineering models equivalent to the models used in the paper. Accounting for all assumptions made in the paper, the simulation routine is executed on the system and the resulting solution of node variables is compared with the values reported in the paper. Results are tabulated Section 5.2. A brief optimization study is subsequently performed on the system and the results are shown to agree with a 2-D sweep of the design space in Section 5.3.

5.1 System Definition

The simple combined cycle configuration is defined as pictured in Figure 5.1. Nodes are numbered for discussion accordingly. Note the use of a pressure reference component in the closed loop as per the discussion of closed loop systems in Section 3.4.

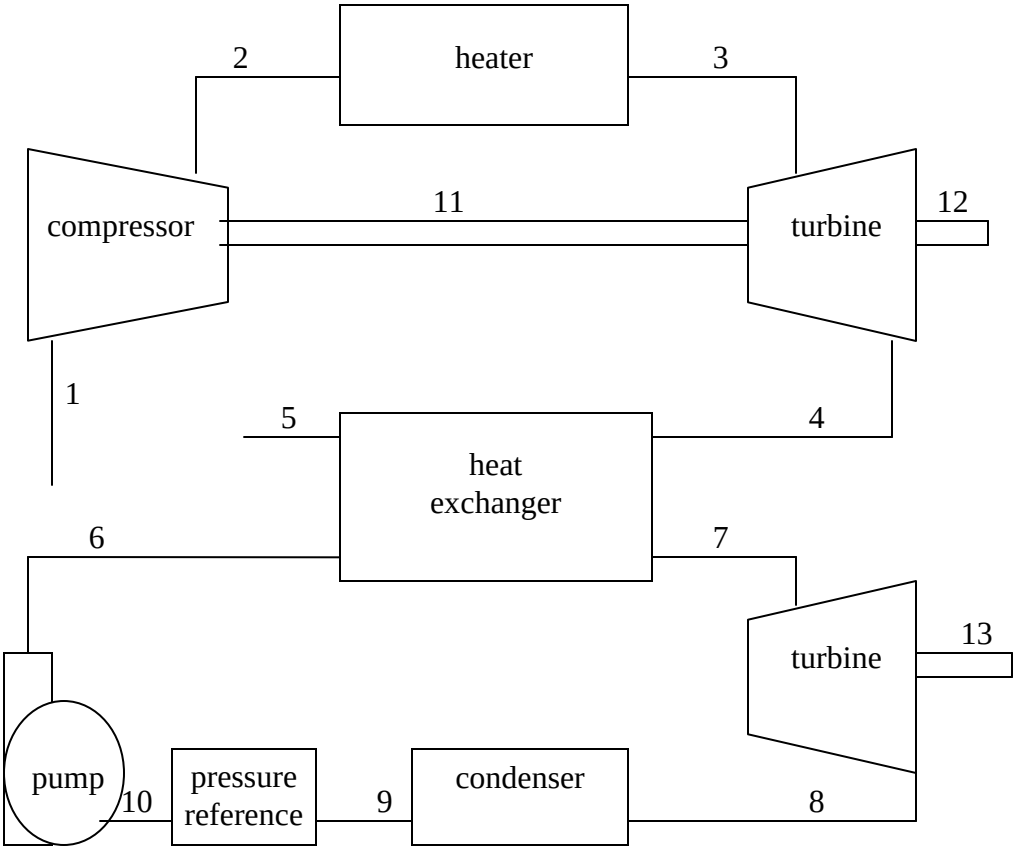

Figure 5.1: Simple Combined Cycle for Case Study I.

According to the paper, the combined cycle shown in Figure 5.1 is subject to the following boundary conditions:

mass_flow1 = 0.0001262 kg/s,

pressure1 = 101300 Pa,

temperature1 = 299.81 K,

pressure5 = 101300 Pa,

angular_velocity13 = 60 s^{-1},

angular_velocity12 = 60 s^{-1},

and indirectly, Pressure Reference = 6551 Pa.

Nodes 1 through 10 are *fluid* domain nodes with node variables *mass_flow*, *pressure* and *specific_enthalpy*. The shaft nodes 11, 12 and 13 are mechanical rotational nodes of domain *mechrot*. They have node variables *angular_velocity* and *torque*. In the paper, power is fixed at these nodes. Since power is not a node variable in the mechanical

rotational domain, *angular_velocity* is fixed at 60 s^{-1} and torque boundary conditions are applied to get the corresponding power using the relation:

power = (*angular_velocity*)(*torque*).

One of the key assumptions made by Wicks is that the system in modeled with respect to a 1 lbm/hr mass flow rate (0.0001262 kg/s) through the gas cycle and assumed to be scalable to whatever level of power is desired from the system. While this assumption may not be true for a real system, the purpose of this case study is merely to validate that the simulation and optimization algorithms perform adequately.

Other technical assumptions made by Wicks are:

System operates at a steady state.

Constant gas side specific heat = 1004.83 J/(kg-K).

No pressure change across components except turbines, pump and compressor.

The following component engineering models are then developed using a set of equations equivalent to the equation set used by Wicks [24]. In this example only, node variables are in italics and user supplied parameters are in bold. Calculated values are in normal text:

For all components except the pressure reference: e(1) = *mass_flow_in – mass_flow_out*

Compressor

e(2) = *mass_flow_in *(enthalpy_out - enthalpy_in)* – **power_in** / **efficiency**

e(3) = *pressure_out - pressure_in****compression ratio**

Heater

e(2) = *mass_flow_in*(enthalpy_out – enthalpy_in)* - **power_in**

e(3) = *pressure_out – pressure_in*

Gas Turbine

e(2) = *mass_flow_in *(enthalpy_in– enthalpy_out)* - **power_out * efficiency**

e(3) = *pressure_in - pressure_out****decompression ratio**

Heat Exchanger

e(3) = *hot_side_mass_flow_in*(hot_side_enthalpy_in – hot_side_enthalpy_out)* - **UA***log_mean_temperature_difference

e(4) = *cold_side_mass_flow_in *(cold_side_enthalpy_in – cold_side_enthalpy_out)* - **UA***log_mean_temperature_difference

e(5) = *hot_side_pressure_in – hot_side_pressure_out*

e(6) = *cold_side_pressure_in – cold_side_pressure_out*

Pump

e(2) = *mass_flow_in *(enthalpy_out – enthalpy_in)* - **power_in / efficiency**

e(3) = *pressure_out - pressure_in****compression ratio**

Steam Turbine

e(2) = *pressure_in - pressure_out****decompression ratio**

e(3) = *mass_flow_in *(enthalpy_in – enthalpy_out)* - **power_out * efficiency**

Condenser

e(2) = *enthalpy_in – enthalpy_out* - enthalpy_of_phase_change

e(3) = *pressure_in – pressure_out*

Pressure Reference

e(1) = *pressure_in* – **reference_pressure**

e(2) = *pressure_out* – **reference_pressure**

e(3) = *enthalpy_in – enthalpy_out*

The system parameters given in the paper are tabulated in Table 5.1.

Table 5.1: Parameters for Case Study I.

Parameter Name	**Value [units]**
Compression Ratio	14 [-]
Gas Turbine Power Out	89.33 [W] per lbm/hr of gas side mass flow
Compressor Power In	49.06 [W] per lbm/hr of gas side mass flow
Heater Power In	106.97 [W] per lbm/hr of gas side mass flow
Heat Exchanger Power Transferred	41.19 [W] per lbm/hr of gas side mass flow
Steam Turbine Power Out	17.07 [W] per lbm/hr of gas side mass flow
Pump Power In	0.23 [W] per lbm/hr of gas side mass flow
Condenser Power Out	27.35 [W] per lbm/hr of gas side mass flow

Components are numbered sequentially as shown in Table 5.2.

Table 5.2 Component Numbers in Case Study I

Component	**Number**
Compressor	1
Heater	2
Gas Turbine	3
Heat Exchanger	4
Pump	5
Steam Turbine	6
Condenser	7
Pressure Reference	8

The user input and execution script for this simulation is posted in its entirety in Appendix A. User inputs follow the conventions put forth in Sections 3.2 and 3.4.

5.2 Simulation

Once the system is defined according to the inputs described above and in Appendix A, a single simulation is run and the operating state of the system is determined and compared with the system operating state in Wicks' paper [29]. Results are tabulated in Table 5.3. Mass flow rates are given in kg/s, temperatures in Kelvin and pressures in Pa. Torques are in Newton-meters and angular velocities in radians per second (s^{-1}) as stated on page v.

Table 5.3 Case Study I Simulation Results

Node Variable	TEPSS Solution	Published Value [24]	% difference
mass_flow1	0.0001262	0.0001263	(fixed)
temperature1	299.8	299.81	(fixed)
pressure1	101300	101300	(fixed)
mass_flow2	0.0001262	0.0001263	0
temperature2	687.14	687.7	0.0814
pressure2	1418200	1418200	0
mass_flow3	0.0001262	0.0001262	0
temperature3	1531.0	1533.2	0.1435
pressure3	1418200	1418200	0
mass_flow4	0.0001262	0.0001262	0
temperature4	827.0	826.8	0.0242
pressure4	101300	101300	0
mass_flow5	0.0001262	0.0001262	0
temperature5	477.1	477.6	0.1047
pressure5	101300	101300	0
mass_flow6	0.00001428	0.00001428	0
temperature6	311.9	311.9	0
pressure6	13782000	13782000	0
mass_flow7	0.00001428	0.00001428	0
temperature7	752.8	755.37	0.3402
pressure7	13782000	13782000	0
mass_flow8	0.00001428	0.00001428	0
temperature8	310.9	310.9	0
pressure8	6551	6551	(fixed)
mass_flow9	0.00001428	0.00001428	0
temperature9	310.9	310.9	0
pressure9	6551	6551	0
mass_flow10	0.00001428	0.00001428	0
temperature10	310.9	310.9	0
pressure10	6551	6551	(fixed)
torque11	0.808	0.808	0
angular_velocity11	60	unpublished	-
torque12	0.6719	unpublished	-
angular_velocity12	60	unpublished	-
torque13	0.2847	unpublished	-
angular_velocity13	60	unpublished	-

Table 5.3 shows strong agreement between the published values and the results obtained from TEPSS. Enthalpy, not temperature, is the third node variable in the 'fluid' domain, but the published values of enthalpy do not correspond to the same reference state as the enthalpies calculated by FluidProp. For this reason, the temperature solutions are compared to the published values instead. Agreement in pressure and temperature on a node is sufficient to establish agreement in enthalpy for a single phase fluid. Values are tabulated for every node variable in the system, even for those that are fixed by boundary conditions (labeled '(fixed)'). All of the node variables in the solution agree with the published values to within 1% or better. All non-temperature values agree to within the user specified tolerance of the simulation algorithm ($1x10^{-8}$). The percent error of node variables agreeing with the published values to this degree is taken to be zero. Temperature values deviate slightly from the published values. These small inconsistencies can be attributed to inconsistencies between the published temperature at a node and the temperature calculated by FluidProp.

Appropriate values for torque are back calculated from the fixed power values in Table 5.2 assuming angular velocity to be fixed at 60 s^{-1} and using the relationship *power = (torque)(angular_velocity)*. Interestingly, the engineering models for all components contain strictly linear relationships describing the pressure change across a component and mass flow through a component. Table 5.3 shows agreement to within the machine's tolerance for these values. This loosely suggests that TEPSS is able to solve linear systems of equations very easily. This simulation was performed on a computer with 3GB of RAM and a dual core Intel® processor. Total time to reach a solution of the 27 unknowns from the initial guess is less than three seconds.

The solution process takes four iterations of Newton's method to solve the system of nonlinear algebraic equations proposed in the component engineering models in Section 5.1 given the initial guess shown in Appendix A. The norm of the system's residual vector holds a value of $1.011x10^{-9}$ at the solution point, indicating that the SNAE has been solved to within the prescribed tolerance of $1x10^{-8}$ (*solver_inputs.eps*).

5.3 Optimization of the Combined Cycle

After demonstrating that TEPSS is capable of simulating the combined cycle system, a brief optimization is performed on the system to determine the optimal fuel rate and heat exchanger size to produce mechanical work at the lowest possible cost per unit of energy. Capital costs and fuel costs are accounted for. All of the original assumptions stated in Section 5.1 still apply in this optimization study, along with additional cost assumptions.

Cost assumptions used for optimization are listed in table 5.3. Some of the values are stated in terms of cost per pound of mass flow per hour through the gas cycle because of the first assumption in Section 5.1. Additionally, a pseudo-constraint is levied on the gas turbine inlet temperature such that the flue gas entering the turbine does not exceed 1700K due to material property constraints. This assumption is also made in the reference paper.

TABLE 5.4 Optimization Cost Assumptions for Case Study I

Parameter	Value
Gas cost	0.025$/kwh
Heat Exchanger Cost	9[$·K/w]UA per lbm/hr of gas side mass flow
System Fixed Cost	$45 per lbm/hr of gas side mass flow
System Lifetime	30 years
Value of Money Over the System Lifetime	Constant

The cost function is then calculated as shown in eqs (5.1 & 5.2).

If T3 < 1700 K Cost = $\frac{(\text{Gas Cost})\cdot(\text{Qin})\cdot(30\text{ yrs})\cdot(8.766\text{ kwh/w-yr})*(0.001\text{kwh/wh})+\text{Fixed Cost + Heat Exchanger Cost}}{\text{Net Mechanical Power Produced [w]}\cdot 8.766\text{ [kwh/w}\cdot\text{yr]}\cdot 30\text{ [yr]}}$

(5.1)

If T3 > 1700 K Cost = $\frac{(\text{Gas Cost})\cdot(\text{Qin})\cdot(30\text{ yrs})\cdot(8.766\text{ kwh/w-yr})+\text{Fixed Cost + Heat Exchanger Cost} + 0.01\cdot(T3-1700)^2}{\text{Net Mechanical Power Produced [w]}\cdot 8.766\text{ [kwh/w}\cdot\text{yr]}\cdot 30\text{ [yr]}}$

(5.2)

where Qin is given in Watts and the other values are given in Table 5.4.

The exterior penalty function is applied in cases where the flue gas temperature entering the gas turbine exceeds 1700K (eq. (5.2)).

Adiabatic heat exchangers are characterized by a heat transfer coefficient *UA*. The rate of heat transfer *Q* is calculated as shown in eq. (5.3). The log mean temperature difference ΔT_{lm} is defined in eq. (5.4):

$$Q = UA\Delta T_{lm} \tag{5.3}$$

$$\Delta T_{lm} = \frac{\Delta \mathrm{T}_{\mathrm{hot\_side}} - \Delta T_{cold\_side}}{\ln(\Delta \mathrm{T}_{\mathrm{hot\_side}} / \Delta \mathrm{T}_{\mathrm{cold\_side}})} \tag{5.4}$$

The power added to the system by the heater and the *UA* parameter of the heat exchanger is varied to determine the most cost effective operating parameters for the system. The system cost per kWh is considered to be the initial system cost (fixed) plus the heat exchanger cost (varies with the value of *UA*) plus the lifetime fuel cost, all divided by the number of kilowatt hours generated over the system lifetime (See eq. (5.1) and code in Appendix A). As the energy added by the heater increases, so too will the fuel cost of the system, the power produced by the system and the gas cycle flue gas temperature. An optimal value will occur when the rate at which mechanical work increases per unit heat added by the heater stop increasing or if the flue gas temperature reaches 1700 K at the heater outlet. As the *UA* value of the heat exchanger is increased, the cost of the system increases and so does the amount of energy transferred to the steam cycle. An optimal *UA* value will occur when the cost of increasing the size of the heat exchanger is equal to the benefit of producing additional power in the steam cycle.

The TEPSS program is used to simulate and optimize the design variables to minimize the cost function, producing the lowest cost per kilowatt hour of work generated. The component parameters are the same as in Table 5.1 with the following exceptions. The parameter *UA* is provided to the heat exchanger instead of *Qhx* and the steam turbine power parameter *wt* is no longer needed as it is calculated based on the change in enthalpy across the component such that the enthalpy of the fluid exiting the turbine is that of a saturated vapor. The component models used are the same as in Section 5.1. Simulation boundary conditions are the same as in Section 5.1.

The optimization routine is executed and the results are reported in Table 5.5. Results are tabulated in terms of the gas cycle mass flow rate in accordance with the first assumption in Section 5.1.

Table 5.5: Combined Cycle Optimization Results

Quantity	**Optimized Value**
Optimal Qin [W]	129.08·gas cycle mass flow rate (lbm/hr)
Optimal UA value [W/K]	2.731·gas cycle mass flow rate (lbm/hr)
Minimum cost [$/kWh]	0.0485

To support the conclusion that the optimal solution obtained by TEPSS is indeed a minimum of the objective function to within the specified tolerance, a two-dimensional sweep of the objective function is performed in the vicinity of the reported minimum. To generate this plot, the design space in question is divided into a 2-dimensional grid (10x10 in this case). The cost function is evaluated for each set of design variables on the grid, resulting in a 2-dimensional array of cost function values. A contour plot is produced from this array using MATLAB's *contourf* function. Figure 5.2 shows a contour plot of the design space surrounding the optimal point.

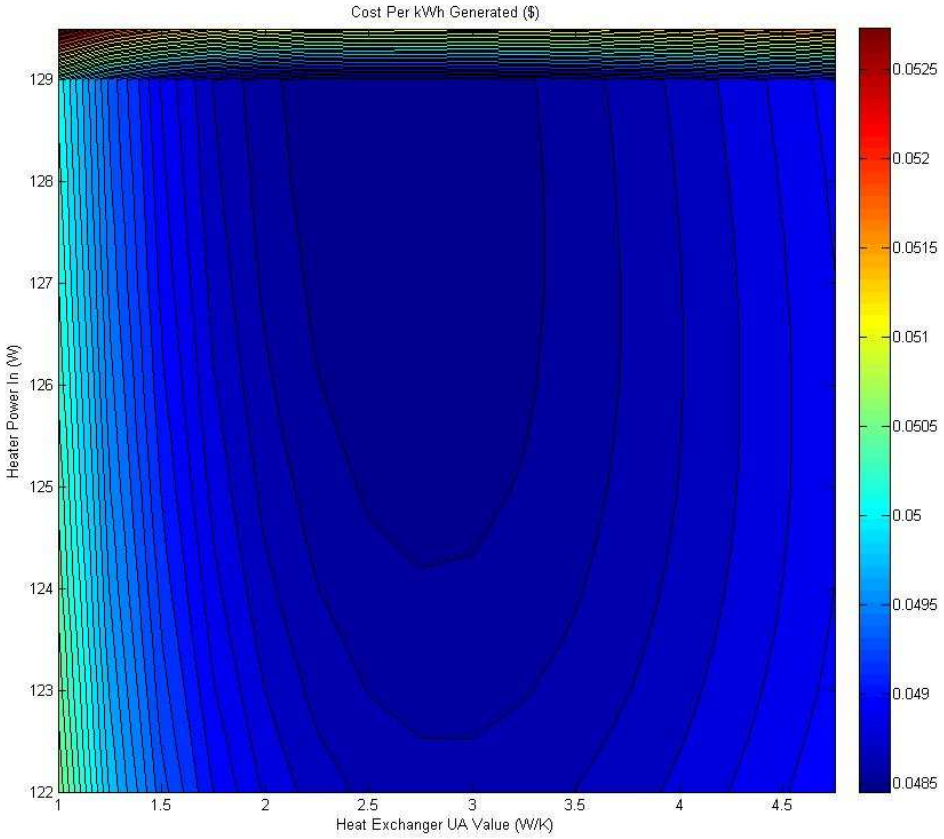

Figure 5.2: Contour Plot of the Cost Function Near the Optimal Point.

Significant insight can be drawn from the contour plot. Most importantly, the figure supports the claim that the solution generated by TEPSS: (2.7 [W/(m-K], 129 [W]) is the minimum of the objective function to within the tolerance of the optimization routine *fmincon*. The effect of the pseudo-constraint on flue gas temperature is also clearly visible at the top of the figure. Further analysis confirms that the flue gas temperature at the optimal point is indeed equal to its maximum value of 1700 K. For a constant heater power, increasing the *UA* from 1 to 5 W/(m-K) causes the cost function to decrease and then increase, illustrating the trade off between additional heat recovery and additional cost. For constant values of *UA*, the cost function generally decreases slowly as the heater power increases in the design space shown until the power reaches just over 129 watts per pound/hr of mass flow. At this point the temperature of the flue gas exceeds the pseudo-constrained maximum of 1700 K.

Chapter 6

Case Study II: Optimization Involving a Thermoelectric Heat Recovery Platform

A hypothetical energy system containing a thermoelectric heat recovery platform is proposed and optimized. The thermoelectric platform has been developed by Andrew Freedman specifically for implementation in TEPSS [30]. The component model is a heat exchanger with thermoelectric modules in between the hot and cold fluid streams. A finite element model is used to calculate the electrical power extracted from the system. Heat exchanger geometry and material properties are all defined explicitly by the user for maximum versatility. In the optimization study, two independent parameters of the heat recovery platform are selected as design variables to minimize the cost per net kilowatt hour of electricity generated over the lifetime of the proposed system.

6.1 Thermoelectric Power Unit Component Class Definition

The thermoelectric power unit component developed for TEPSS has been specifically designed to take advantage of the versatility of the platform and to be versatile itself. For this reason, the component requires the user to define many more parameters than the components discussed elsewhere in this document where simple first principles models were used. For the sake of clarity, the *parameters.tepowerunit* structure for this component is broken into 5 additional subfields *module*, *unit*, *fins*, *cost* and *options*. Each of which have related subfields discussed in [30] and listed in Appendix A within the user input and execution file template code for this case study. Use of the *tepowerunit* class of component requires a working installation of FluidProp (see Section 4.1 for all system and software requirements).

The thermoelectric power unit component class defined in tepowerunit.m is essentially a more sophisticated heat exchanger component model with numerous additional options and a subroutine for the calculation of thermoelectric power generation and other thermoelectric effects. The component allows a user to define a variety of heat exchangers that have the basic geometry of two parallel rectangular ducts separated by a two-dimensional array of thermoelectric modules or an array of modules sandwiched in between an isothermal surface and a rectangular duct. For the case of two ducts, parallel and counter flow engineering models are available. Heat transfer fins in the system ducts may be either rectangular or cylindrical and in line or staggered, creating four possible geometric paradigms for fin configurations. Additional parameters such as fin thickness/diameter, fin length, fin density and material conductivity are all customizable by the user. For the case of two ducts, geometry in each duct can be independent of the other. Thermoelectric module dimensions and properties are supplied along with the number of modules in the power unit and their spacing. Insulation of a user-prescribed thermal resistivity fills the gaps where no modules exist in the heat transfer plane. Ceramic wafers common on thermoelectric modules can also be added. Different paradigms are available for definition of the module behavior. Much more information on the functionality and options of this component is available in [30].

The component's ducts are discretized into a finite number of elements (zones). Zones are all in thermal series with one another [31]. A log mean temperature difference relationship is used to calculate the temperature change across each zone. For the purposes of module level calculations, the hot and cold sides of modules are assumed to be isothermal within a zone. For this reason, greater accuracy can be achieved in thermoelectric calculations at the expense of computational power by increasing the number of zones and/or decreasing the number of modules in thermal series in each zone. Within the component model, MATLAB's *fsolve* nonlinear equation solving routine is used to solve a common system of equations in each zone. These equations account for phenomena such as temperature and pressure drop across each zone, environmental heat losses, thermal and electrical contact resistances, two dimensional heat spreading and thermoelectric-induced phenomena. The modules are assumed to operate at their peak power for the temperature difference.

6.2 System Definition

The thermal system simulated in this case study is illustrated in Figure 6.1. The system is similar to a simple Brayton Cycle with heat recovery. The heat exchanger used in the system is replaced with a thermoelectric heat recovery platform so that some of the heat recovered is converted directly into electricity.

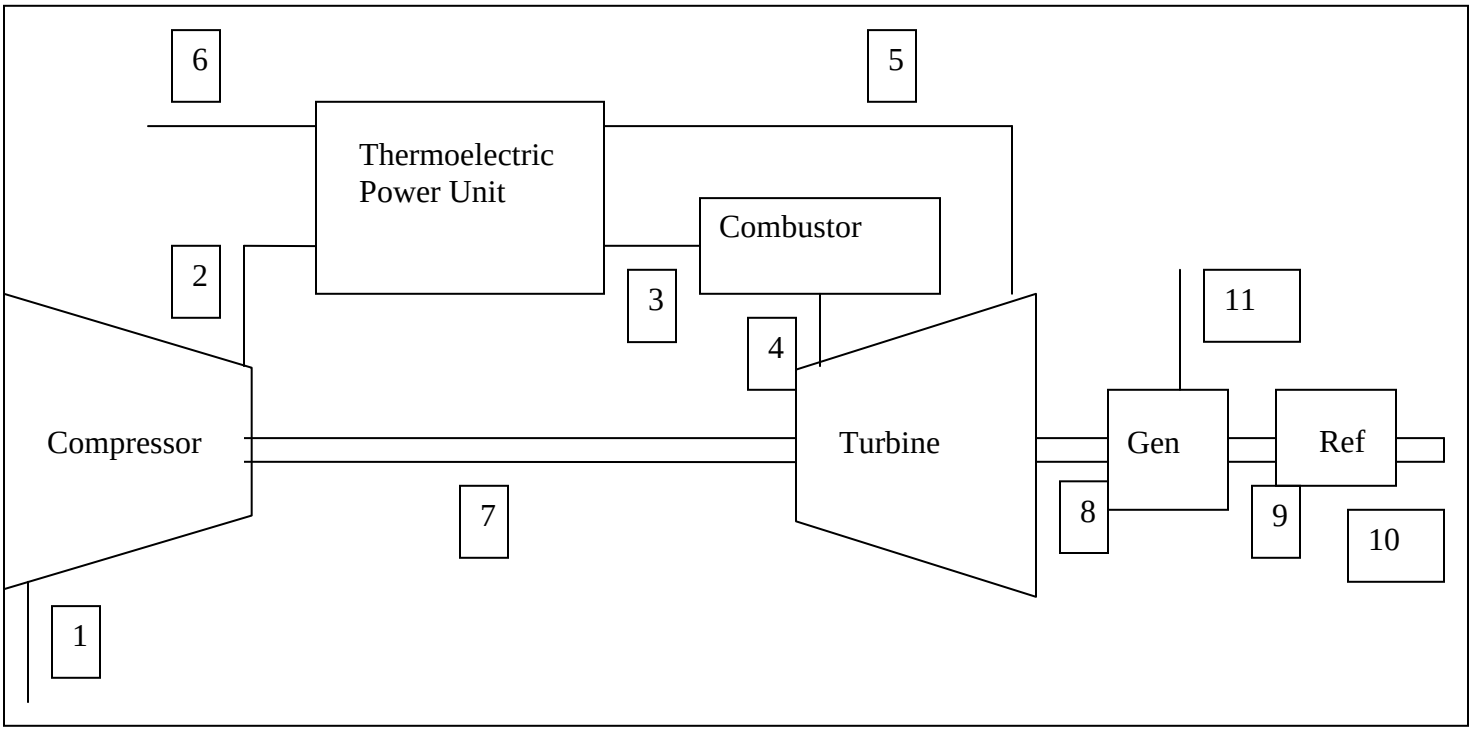

Figure 6.1: Case Study II System Illustration.

Nodes are numbered according to Figure 6.1, nodes 1-6 are of the fluid domain, nodes 7-10 are of the mechanical rotational domain and node 11 is electrical. As for system boundary conditions, pressure and temperature are fixed at node 1 to 101300 Pa and 300 K, pressure at node 6 is fixed to 101300 Pa and rotational velocity at node 10 is fixed to zero. The component labeled ‘Ref’ is a new component across which a fixed difference in angular velocity exists. The component labeled ‘Gen’ is an ideal AC generator. The system is designed to mimic the operation and performance of an aeroderivative gas turbine power cycle. These types of systems are common in remote power applications. The net power of such systems typically falls in the range of 20 to 40 MW.

The following technical assumptions are made prior to formulating the engineering model of the system. The system performs at a steady state, components other than the thermoelectric power unit are isentropic (adiabatic and reversible). The power unit is anisentropic because environmental losses are calculated and accounted for within the component. The fluids in the system behave as ideal gasses and shaft work is geared down to 60 s^{-1} using an ideal gear box inside the turbine and geared up similarly within the compressor.

Since there is no cooling system present on the gas turbine component, the maximum temperature of the flue gas is limited to 1700 K, as in Chapter 5, due to material property constraints. The system efficiency of this type of system will increase with the gas turbine inlet temperature. As the parameters of the power unit are adjusted, the heat recovered by the component will vary. The combustor will add the remainder of the heat necessary after preheat to get the temperature of the flue gas at the current mass flow rate to 1700 K. Mass flow rate is a function of the power supplied to the compressor (8MW constant) and the pressure drop across the thermoelectric power unit component, which is calculated from the geometry of the component. The air fuel mixture in the system is assumed to behave similar to an ideal gas comprised of 75.5% diatomic nitrogen, 19% diatomic oxygen and 5.5% methane gas by mass. Ideal gas law relationships resident in FluidProp's GasMix database are used to carry out all fluid property calculations at the component and node level. Exhaust gas is assumed to behave similarly to the air/fuel mixture [28].

6.3 Optimization of Selected Thermoelectric Power Unit Parameters

A set of two independent design variables are selected from the parameters of the thermoelectric power unit to be optimized. A cost function is formulated to calculate the cost (fixed plus fuel) per kilowatt hour of net electrical energy output by the system. One of the fixed costs is the cost of the thermoelectric power unit; this cost varies with the size of the power unit and the number of modules in it. As a result, the fixed cost is a function of one or more of the design variables. The amount of fuel used also varies with

the heat exchanger parameters because as mentioned above, the amount of fuel used is proportional to the amount of heat required to raise the temperature of the air/fuel mixture from its preheated state exiting the power unit to 1700 K. If a larger, more costly power unit is used, more heat can be recovered, saving fuel. Pressure losses experienced across the heat exchanger will parasitically detract from the amount of power generated by the turbine. There is likely to be a point at which the cost associated with the additional pressure loss due to lengthening the power unit is equal to the benefit of the fuel cost savings associated with additional heat recovery and the additional thermoelectric power generation. For this reason, the number of thermoelectric modules in thermal series in each of the twenty geometrically congruent zones is chosen as a design variable.

Rectangular in-line fins are chosen to extend the surface area inside the power unit. Zone width is fixed (1200 4cm modules wide), so adding additional fins will decrease the cross sectional area through which the flue gas may flow. The resulting increase in velocity through the power unit will cause the pressure drop across the unit to rise and the turbine power to drop. There should be a point at which the benefit of added heat recovery due to increased surface area is equal to the cost of decreased turbine power, so the number of fins spanning the width of the power unit is selected as a design variable. Appropriate fin thicknesses and lengths are chosen and fixed to yield reasonable fin performance in the design space chosen.

According to this qualitative analysis, and the assumption that the material costs are linearly proportional to the amount of material used, the design variables should only have one break even point for which additional cost = additional benefit over the range of their feasible values (all positive numbers for which the fins do not completely block the duct). This guarantees a convex design space, within which a single minimum will exist either on the boundary or at some interior point.

The parameters assigned to the system are shown in Table 6.1. The values listed for the parameters ‘number of fins’ and ‘number of modules in thermal series’ are the initial guesses of the two design variables. All other parameters are fixed to the value shown throughout the optimization routine. Units are all in kilograms, meters, Kelvins, seconds, coulombs, radians, present day US dollars and their combinations as per the TEPSS convention stated on page v.

Table 6.1 System Parameters for Case Study II.

Component 1: Compressor

Variable	Parameter	Value
parameters.compressor.CR	Compression ratio	35
parameters.compressor.wc	Work done on the fluid by the compressor	6,400,000 [W]
parameters.compressor.shaftspeed	Angular velocity of input shaft	60 [s^{-1}]
Parameters.compressor.efficiency	Compressor Efficiency	0.8
parameters.compressor.direction	Expected inlet/outlet locations	`solver_inputs.cn map(1,2:4);`

Component 2: Thermoelectric Power Unit

Variable	Parameter	Value
parameters.tepowerunit.fins.t_h	Hot side fin thickness	0.001 [m]
parameters.tepowerunit.fins.l_h	Hot side fin length	0.01 [m]
parameters.tepowerunit.fins.num_h	Hot side number of fins	7200 (guess)
parameters.tepowerunit.fins.k_h	Hot side fin thermal conductivity	250 [W/m-K]
parameters.tepowerunit.fins.base_t_h	Thickness of the base of the fin array (hot side)	0.0075 [m]
parameters.tepowerunit.fins.t_c	Cold side fin thickness	0.001 [m]
parameters.tepowerunit.fins.l_c	Cold-side fin length	0.01 [m]
parameters.tepowerunit.fins.num_c	Cold side number of fins	7200 (guess)
parameters.tepowerunit.fins.k_c	Cold side fin thermal conductivity	250 [W/m-K]
parameters.tepowerunit.fins.base_t_c	Thickness of the base of the fin array on the cold side	0.0075 [m]
parameters.tepowerunit.unit.series	Number of modules in thermal series (per zone)	10 (guess)
parameters.tepowerunit.unit.parallel	Number of modules in thermal parallel (# across the width of the power unit)	1200
parameters.tepowerunit.unit.num=20;	Number of in line finite elements (zones)	20
parameters.tepowerunit.unit.insul_k	Thermal conductivity of insulation	0.05 [W/m-K]
parameters.tepowerunit.unit.zone_to_mod_area_ratio	Ratio of total heat exchanger flux area to module area	1.01

Variable	Parameter	Value
parameters.tepowerunit.unit.therm_contact_res	Thermal contact resistance between power unit and module	0.000001 [K/W]
parameters.tepowerunit.unit.uvalue	Overall heat transfer coefficient to environment	0.01 [W/K]
parameters.tepowerunit.unit.shell_t	Thickness of power unit walls	1 [m]
parameters.tepowerunit.unit.shell_k	Conductivity of power unit walls	0.000001 [m]
parameters.tepowerunit.unit.envir_temp	Envoronmental temperature	300 [K]
parameters.tepowerunit.module.rho_p	Electrical resistivity of p-type semiconductor	0.000004 [Ω-m]
parameters.tepowerunit.module.alpha_p	Seebeck coefficient of p-type semiconductor	0.0004 [V/K]
parameters.tepowerunit.module.k_p=1	Thermal conductivity of p-type semiconductor	1 [W/m-K]
parameters.tepowerunit.module.l_p	Length of p-type TE leg	0.005 [m]
parameters.tepowerunit.module.area_p	Leg cross sectional area for p-type semiconductor	$(0.001397 \text{ [m]})^2$
parameters.tepowerunit.module.rho_n	Electrical resistivity of n-type semiconductor	0.000014 [Ω-m]
parameters.tepowerunit.module.alpha_n	Seebeck coefficient of n-type semiconductor	0 (all Seebeck given to p-type) [V/K]
parameters.tepowerunit.module.k_n=1;	Thermal conductivity of n-type semiconductor	1 [W/m-K]
parameters.tepowerunit.module.l_n	Length of n-type TE legs	0.005 [m]
parameters.tepowerunit.module.area_n	Cross sectional area of n-type legs	$(0.001397 \text{ [m]})^2$
parameters.tepowerunit.module.a_ratio	Ratio of module area to semiconductor cross sectional area in a module	2.7
parameters.tepowerunit.module.num	Number of leg pairs in a module	127
parameters.tepowerunit.module.l_cer	Ceramic thickness	0.0008 [m]
parameters.tepowerunit.module.k_cer	Ceramic thermal conductivity	25 [W/m-K]
parameters.tepowerunit.module.contact_resist	Contact resistance in the module	$3x10^{-9}$ [Ω/m^2]
parameters.tepowerunit.cost.specific.module_p_leg_material	Cost per cubic meter of p-type material	0 [$/m^3]
parameters.tepowerunit.cost.specific.module_n_leg_material	Cost per cubic meter of n-type material	$8x10^7$ [$/m^3]
parameters.tepowerunit.cost.specific.module_ceramic	Cost per cubic meter of ceramic material	0 [$/m^3]
parameters.tepowerunit.cost.specific.cost_per_leg_pair	Additional cost per leg pair	0 [$]
parameters.tepowerunit.cost.fixed.module_manufac	Additional manufacturing cost	0 [$]

Variable	Parameter	Value
parameters.tepowerunit.cost.specific.fin_material	Cost per cubic meter of fin material	40000 [$/m^3]
Parameters.tepowerunit.cost.fixed.fin_manufac_h	Additional fin manufacturing cost (hot side)	0 [$]
Parameters.tepowerunit.cost.fixed.fin_manufac_c	Additional fin manufacturing cost (cold side)	0 [$]
Parameters.tepowerunit.cost.specific.insulation	Cost per cubic meter of insulation	0 [$/m^3]
Parameters.tepowerunit.cost.specific.cost_per_zone_area	Cost per square meter of heat transfer area	0 [$/m^2]
parameters.tepowerunit.cost.fixed.other	Miscellaneous fixed cost	0 [$]
parameters.tepowerunit.cost.fixed.assembly	Assembly cost	1 [$]
parameters.tepowerunit.direction	Expected inlet/outlet locations	`solver_inputs.cnmap(2,2:5);`
parameters.tepowerunit.options	Power unit settings	`{'option2','','straightfins_aligned','straightfins_aligned'};`
Parameters.tepowerunit.cost.specific.cost_per_zone_area	Cost per square meter of heat transfer area	0 [$/m^2]
parameters.tepowerunit.cost.fixed.other	Miscellaneous fixed cost	0 [$]
parameters.tepowerunit.cost.fixed.assembly	Assembly cost	1 [$]
parameters.tepowerunit.direction	Expected inlet/outlet locations	`solver_inputs.cnmap(2,2:5);`
parameters.tepowerunit.options	Power unit settings	`{'option2','','straightfins_aligned','straightfins_aligned'};`

Component 3: Heater

Variable	Parameter	Value
parameters.heater.toutmax	Max heater outlet temperature	1700 [K]
parameters.heater.LHV	Lower heating value of fuel (Methane)	50x10^6 [J/kg]
parameters.heater.stoichafr	Air-methane stoichiometric air fuel ratio	17.2
parameters.heater.direction	Expected inlet/outlet locations	`solver_inputs.cnmap(3,2:3);`

Component 4: Turbine

Variable	Parameter	Value
parameters.turbine.efficiency	Turbine efficiency	0.85
parameters.turbine.shaftspeed	Turbine output shaft angular velocity	60 [s^{-1}]
parameters.turbine.direction	Expected inlet/outlet locations	`solver_inputs.cnmap(4,2:5);`

Component 5: Generator

Variable	Parameter	Value
parameters.generator.damperresistance	Damping constant	5555 [N-m-s]
parameters.generator.Kv	Voltage constant	5555 [V-s]
parameters.generator.direction	Expected inlet/outlet locations	`solver_inputs.cnmap(5,2:4);`

Component 6: Rotational Reference

Variable	Parameter	Value
parameters.rotationalreference.refspeed	Rotational velocity of reference node	0 [s^{-1}]
parameters.rotationalreference.direction	Expected inlet/outlet locations	`solver_inputs.cnmap(6,2);`

These and the remaining user inputs – the *solver_inputs* structure and optimization inputs – are available for reference in the sample code of the user inputs and execution file for this case study provided in Appendix A. Observe that the module dimensions aren't explicitly defined by the user, instead, the leg area, the number of leg pairs and an area ratio are used to calculate the dimensions. For this set of inputs, the module size is 4cm by 4cm, which is typical for a commercially available module. Also, the fin height is fixed at 1cm the fin thickness is fixed at 1mm and the compressor draws a fixed 8 MW of the turbine's gross power for all simulations.

The cost function being optimized is formulated similarly to the one in Case Study I (Chapter 5). The cost metric used to determine feasibility is the system life cycle

cost (fixed plus fuel) per kilowatt-hour of net electricity produced over the system lifetime. The input *cost_function_def* is defined in accordance with eq. (3.1) as:

A = '*component_cost.cost*(1)' %fixed cost
B = '*component_cost.power*(2) * 0.025 * 8.766' %fuel cost
D = '*component cost.power*(1) * 8.766' %net power generated
t = 30 %system lifetime in years.

all other cost inputs in *cost_function_def* are zero. The resulting cost function calculation looks exactly like eq. (5.1) except without the heat exchanger cost.

Relevant cost factors tabulated in Table 6.1 are $8x10^7$ \$/m^3 of thermoelectric material and $4x10^4$ \$/m^3 of fin material. These are derived from typical material costs and will be integral in determining the optimal system configuration. The assumptions are made that the thermoelectric module cost is dominated by the amount of semiconductor material in the module and that the total fin and module costs are directly proportional to the amount of material used. Economic assumptions are made similar to the ones in Chapter 5; they are tabulated in Table 6.2

Table 6.2 Economic Assumptions for Case Study II.

Parameter	**Value**
Fuel cost	0.025\$/kwh (constant)
System Lifetime	30 years
Value of Money Over the System Lifetime	Constant

An error is experienced when running the optimization routine with fewer than 20 zones and more than 2 modules in thermal series per zone. This is presumably because there are not enough finite elements in the power unit to obtain an accurate solution for the pressure and enthalpy change across each side of the power unit. The assumption that hot and cold side temperatures are isothermal within a zone becomes invalid if large zones are used. The solution is to increase the number of zones while keeping the total size of the power unit the same. Increasing the number of zones from 1 to 20 increases

the simulation time dramatically. It takes about twenty times as long to reach a steady state solution with 20 zones as it does for a single zone.

After running the lengthy optimization process, which took several hours, a solution to the optimization problem is obtained and tabulated in Table 6.3. For presentation and discussion, the optimal number of fins is converted into a fin density, which is a better metric for comparison to other systems. Also, the number of modules in thermal series per zone is multiplied by the number of zones (20) and rounded to the nearest integer to show the total length of the heat exchanger in modules.

Table 6.3: Case Study II Optimization Results

Parameter	**Optimal (Unit)**	**Alternative Parameter**	**Converted Optimal (Unit)**
Optimal number of fins	7200 (fins)	Fins per 4 cm module	6.03 (fins/module)
Optimal number of 4cm x 4cm modules in thermal series per zone	4.56 (modules/zone)	Total # of TE modules in thermal series	91(modules)
Objective function minimum value	0.0465 ($/kWh)		

To validate this result, a 2-dimensional contour plot is generated illustrating the values of the objective function in the vicinity of the reported solution. Figure 6.2 shows this contour plot with respect to fin density and total number of modules in thermal series. The plot is generated by the same process as Figure 5.2 in Section 5.3.

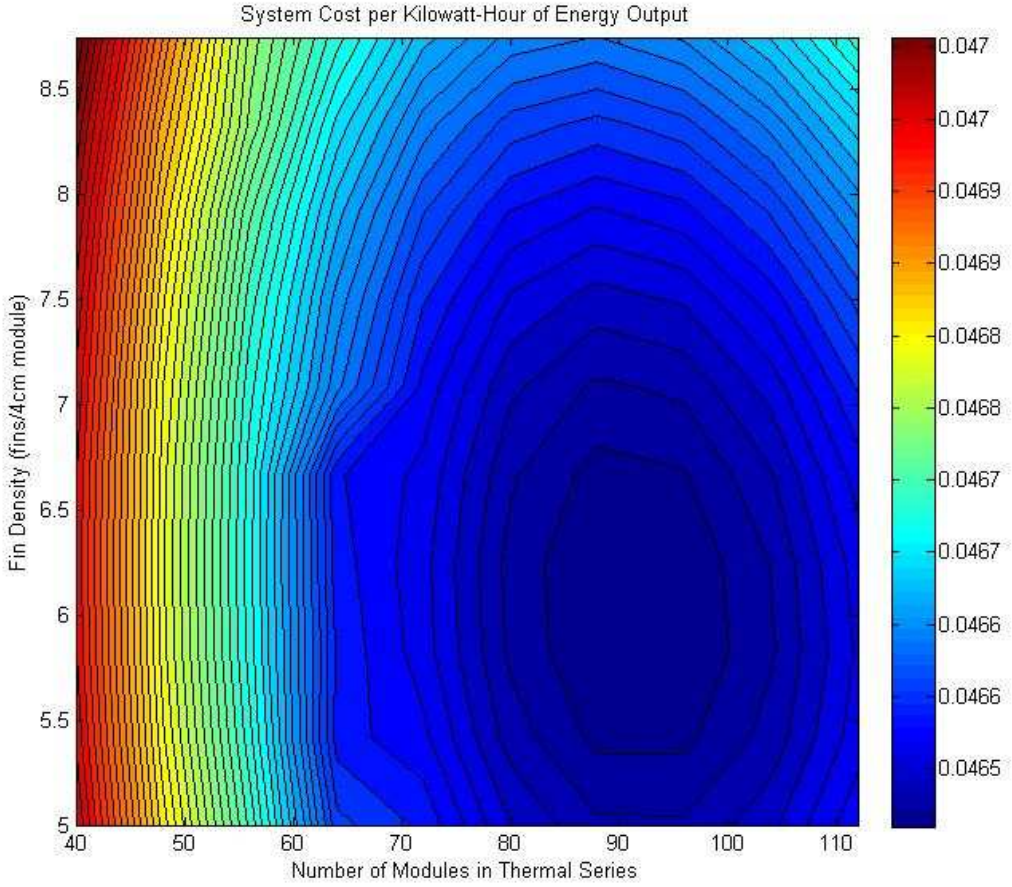

Figure 6.2 System Cost per Kilowatt-Hour of Energy Output

After generating Figure 6.2, it became apparent that the cost function is not particularly sensitive to the design variables in the vicinity of the objective function. It would be nice to compare the percentage of additional cost per kWh of the system studied compared with a base case. At this point, a single simulation is run without the thermoelectric power unit present (simple Brayton cycle, no regeneration), using all other parameters from Table 6.1. The cost per kilowatt hour of this base case is found to be 0.04834 $/kWh. The percent saved by using the heat recovery platform is displayed in the contour plot in Figure 6.3. The optimal (maximum) savings calculated from the

optimization results is a savings of 3.806%. Percent savings is calculated as

$$\%savings = 100 \times (1 - \frac{cost\_per\_kWh}{0.04834}).$$

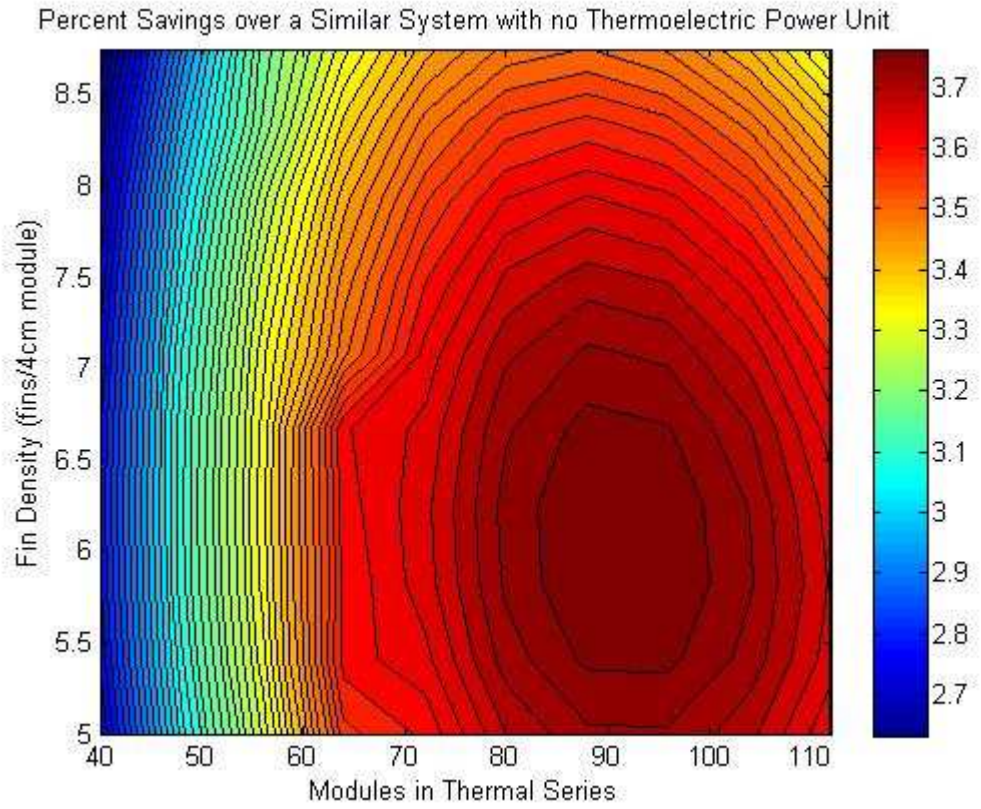

Figure 6.3: Percent Savings over a System with no Thermoelectric Power Unit

Figure 6.3 asserts that for all fin densities and module numbers in the design space, a net savings will exist over a system for which there is no heat recovery or thermoelectric generation.

While the optimization routine finds the optimal solution, one or more system parameters that are not fixed such as overall efficiency or heat recovery may fall out of the desired range. Additional information about the optimal system is tabulated in Table 6.4 to show that no parameters seem exceedingly large or small for the type of system being simulated. The parameters listed fall within a reasonable range for an aeroderivative gas generator.

Table 6.4: Additional Information Regarding the Optimal System

Parameter	Value
Overall Thermal Efficiency	0.537
Turbine Net Power Output	22.73 MW
TE Power Generated	243.2 KW
Heat Recovered	2.900 MW
100·(TE Power / Total Net Power)	1.06%
Fin Array Efficiency	0.926

Keep in mind that the purpose of this case study is to validate that the optimization algorithm performs as expected when used on a system containing a thermoelectric heat recovery platform. While efforts are made to make this case study realistic, it is not intended to be a thorough feasibility study for a real system. The assumptions made and component models used present a simplified situation in which to gather evidence supporting the validity of the optimization solution.

Chapter 7

Concluding Remarks

7.1 Summary of Results

TEPSS is intended to be a versatile software tool for simulation and optimization of any system containing a thermoelectric heat recovery platform. By modifying Newton's method to utilize numerical derivatives, a simulation tool was developed that can find a solution to a wide range of systems of equations provided by the user so long as it is solvable. This sets TEPSS apart from the works published to date on thermoelectric heat recovery that have tended to focus on modeling narrow applications of thermoelectric heat recovery platforms. The modular nature of component models in TEPSS allows reusability in different system configurations with little or no changes to the component model required by the user and little down time in between simulations. These features give TEPSS the versatility needed to be useful in finding feasible applications of thermoelectric heat recovery. As next generation thermoelectric materials become available and manufacturing costs fall, TEPSS can be used to determine system feasibility as technological advances are made.

Section 5.2 compares the published results of a combined cycle simulation with the results generated by TEPSS. The degree of agreement between the data sets strongly supports the case that TEPSS' simulation algorithm is valid. Several special exceptions to the basic algorithm are made to maximize the versatility of the simulation platform. Most notably, the use of FluidProp to allow changes in fluid composition, mixtures of fluids, and fluid phase changes with little effort by the user has made it possible to quickly and easily simulate many types of thermodynamic systems. System checks for solvability provide the user with feedback in the event that a system of equations is determined to be unsolvable and the added ability to track the direction of through variables round out the changes made to the basic implementation of Newton's Method to solve a system of equations.

The optimization shell of TEPSS leverages MATLAB's *fmincon* function to determine the set of design variables that minimizes the specified objective function. This routine carries out minimization of a nonlinear objective function in the user specified design space. TEPSS was applied to two different case studies and optimal solutions were obtained. The optimal solutions were checked by sweeping the design space and generating contour plots that support the conclusion that the optimization routine was able to effectively minimize the cost function (Figures 5.2 and 6.2). In addition to the core optimization algorithm, SUMT methods are utilized to prevent the values of physical quantities from exceeding user defined realistic limits without directly adding any constraints to the objective function. For discrete design variables, the function *fminconset* is run over the top of the core routine to isolate the optimal solution in order to respect the discrete nature of one or more of the design variables. This approach only works if the problem can be solved as though all design variables are continuous first (see future work in Section 7.3). And finally, the optimization routine will abort a simulation but not optimization if the components indicate that the system is unsolvable or unrealistic. In such cases the cost function is given a high value so that the minimization routine will return to feasible space and optimization can continue.

The simulation shell of TEPSS uses an adaptation of Newton's Method developed from scratch to solve the system of nonlinear algebraic equations presented in the component models of a system. Since the equations in the components are not known prior to the start of a simulation, their derivatives are not explicitly known. Consequently, a numerical differentiation routine is employed that uses the centered difference method to calculate the first partial derivative of each equation with respect to each design variable. These values are used to produce an approximation of the Jacobian Matrix for use in Newton's Method.

7.2 Contributions to the Field

The TEPSS platform provides the necessary framework for rapidly and cost effectively simulating and optimizing conceptual energy systems. It allows users to search for feasible applications of thermoelectric heat recovery. The modularity of the component class definition files provides a framework that minimizes the time the user

spends defining each simulation. Components can be used in one system simulation and then in another system without having to change the code in the component class definition file.

The platform is expandable to allow users to develop new components and domains and employ component models that account for the complexities of non-ideal components. TEPSS is capable of handling components that employ finite element models and nodes that contain multiphase fluids and mixtures of fluids.

7.3 Future Work

Upon examining the results of Case Study II (Chapter 6) it became apparent that improvements could be made. While the system with the thermoelectric heat recovery platform outperforms the traditional Brayton Cycle base case without regeneration, it probably would not outperform a similar system with a normal heat exchanger. The reason for this is that a good thermoelectric generator has a high thermal resistance, which would reduce the amount of heat recovery unless a larger and more costly heat recovery unit is used. Restrictions on time and computational power and the lack of a sufficient heat exchanger model precluded this study from taking place. In Case Study II, the heat recovered by the power unit and consequent fuel savings dominate the cost function. An opportunity for a research publication exists if TEPSS could be used to simulate a realistic regenerative Brayton Cycle with both a traditional heat exchanger (for maximum heat recovery) and a thermoelectric heat recovery platform, optimized to maximize power output per unit of additional cost. Such a configuration may produce interesting results pertaining to the optimal thermoelectric module geometry, which is often overlooked in published optimization studies on thermoelectric systems.

In cases where the system is determined to be unsolvable before simulation begins because of a user implemented check within a component, the simulation is skipped and the cost function is set to a large fixed value. This results in a slope discontinuity and the loss of derivative information, which could cause the optimization algorithm to fail. While it is a quick fix to prevent the simulation from failing, a more robust approach would be to have the user implement a continuous penalty function into the checks. This way derivative information could be maintained and the optimization algorithm could be

steered back into feasible space. Such an approach could mimic the interior penalty function described by Vanderplaats [19]. The user defined checks would be considerably more complicated than the binary solvable/unsolvable system approach that is currently employed, so the old system could be kept in tandem for users who do not wish to add unnecessary complexity to their component models.

Solving mixed integer nonlinear problems (MINLP) adds another level of complexity to the already complicated constrained nonlinear optimization. Mixed integer problems occur when one or more design variables is discrete by nature. The current solution method involves solving the problem continuously first, which may not always be possible. Consider a multistage turbine; the number of stages in the turbine is limited to integer values. No engineering model exists for turbines with a non-integer number of stages. If the user wishes for the number of stages to be a design variable in the current version of TEPSS, the number of stages will have to be rounded, disrupting the derivative information and making it difficult to reach a solution. Additional optimization algorithms could be added to TEPSS in the future specifically for solving MINLPs.

During the development of TEPSS, little attention is given to evolutionary algorithms as equation solving or optimization tools. Discussed in Chapter 2, they are generally more computationally intensive algorithms based on natural phenomena. There are some advantages to using evolutionary algorithms if the computational power required is not prohibitive. In non-convex design spaces evolutionary algorithms have a better chance of finding a global minimum without a priori knowledge of its location because they are capable of searching the design space at multiple points at once. While Newton's Method has been reliable thus far, there are a host of known scenarios in which the method will fail. Having a second algorithm available could help prevent such failures.

Many physical systems are loosely coupled. That is, most equations in the engineering model contain only a fraction of all the dependent variables in the system. By extension, most partial derivatives are zero. One way to conserve computing power is by switching from a dense matrix algebra paradigm to a sparse one. This could make calculation of matrix inverses (namely the Jacobian) significantly faster.

If a generic algorithm could be devised and implemented to calculate the gradient of the objective function with respect to the design variables, then *fmincon*'s *trust region reflective* method could be used. It uses sparse linear algebra, which could save a lot of time in large systems with a lot of unknown node variables and design variables.

Finally, the cost function used as a scalar metric of feasibility could be reworked to account for the time value of money if desired. Things like fuel cost inflation, financing costs, monetary inflation and rate of return on investment could be accounted for in such a cost function.

As promised, TEPSS has been shown in its current form to be capable of simulating and optimizing general energy systems. The reusable nature of component class definition files means that only the user inputs need to be changed to switch from simulating one system concept to another. While there is still considerable room for improvement, TEPSS currently provides a significant contribution to the field of thermoelectric system modeling.

References

[1] T. Kajikawa, M. Ozaki, K. Yamaguchi, H. Obara, " Progress of Development for Advanced Thermoelectric Conversion Systems." *2005 International Conference on Thermoelectrics.* pp. 147-154. 2005.

[2] D. T. Crane, G. S. Jackson, “Optimization of cross flow heat exchangers for thermoelectric waste heat recovery”, *Energy Conversion and Management* 2004, 45, (9), 1565-82.

[3] Bethancourt, A.; Echigo, R.; Yoshida, H., Thermoelectric conversion analysis in a counter-flow heat exchanger. *AIP Conference Proceedings* 1995, 316, 299-304.

[4] J. Yu, H. Zhao, A numerical model for thermoelectric generator with the parallel-plate heat exchanger, *Journal of Power Sources* (2007).

[5] E. E. Antonova and D. C. Looman, “Finite elements for thermoelectric device analysis in ansys,” *2005 24th International Conference on Thermoelectrics (ICT)* , IEEE, 2005.

[6] S. Lineykin, S. Ben-Yaakov, " Modeling and analysis of thermoelectric modules," *IEEE Transactions on Industry Applications*, vol. 43, no. 2, pp. 505-512, March, 2007.

[7] E. J. Sandoz-Roszado, “Investigation and development of advanced models of thermoelectric generators for power generation applications”. M.S. thesis, Rochester Institute of Technology, Rochester, NY, USA, 2009.

[8] K. Qiu, A.C.S. Hayden, Development of a Thermoelectric Self-Powered Residential Heating System, *Journal of Power Sources* (2007).

[9] T. Kajikawa, “Thermoelectric power generation systems recovering heat from combustible solid waste in Japan,” *Proceedings of the 1996 15th International Conference on Thermoelectrics*, ICT'96, Pasadena, CA, USA, 1996; Pasadena, CA, USA, 1996; pp 343-351.

[10] T. J. Hendricks, J. A. Lustbader, “Advanced thermoelectric power system investigations for light-duty and heavy duty applications: part 1,” *21st International Conference on Thermoelectrics*, 2002; pp 381-386.

[11] J. Manninen, X. Zhu, “Thermodynamic analysis and mathematical optimisation of power plants,” *Computers & Chemical Engineering*, Volume 22, Supplement 1, European Symposium on Computer Aided Process Engineering-8, 15 March 1998, Pages S537-S544,

[12] D. N. Grekas, C. A. Frangopoulos, “Automatic synthesis of mathematical models using graph theory for optimisation of thermal energy systems”. *Energy Conversion and Management* Volume 48, Issue 11, November 2007, Pages 2818-2826.

[13] K. Walter. “A Quantum Contribution to Technology”. *Science & Technology Review*, 2007.

[14] L. S. Mason., “Realistic specific power expectations for advanced radioisotope power systems,” *Proceedings of the 4th International Energy Conversion Engineering Conference* (IECEC–2006), San Diego, CA, June 26–29, 2006.

[15] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen; K. A. Houston. “Object-oriented analysis and design with applications,” 3ed, Addison-Wesley Professional. 2007.

[16] R. Stevens “NYSERDA PON1190” proposal unpublished, 2008

[17] D. Borisevich, V. G. Potemkin, S. P. Strunkov, H. G. Wood, “Global methods for solving systems of nonlinear algebraic equations”, *Computers & Mathematics with Applications,* Volume 40, Issues 8-9, October-November 2000, Pages 1015-1025.

[18] K. Roach. “Symbolic-numeric nonlinear equation solving”. Department of Computer Science, University of Waterloo Waterloo, Ontario, Canada N2L 3G1

[19] G. N. Vanderplaats, *Multidiscipline Design Optimization*, Vanderplaats R&D, Inc., 2007.

[20] Y. Moa, H. Liu, Q. Wang. ”Conjugate direction particle swarm optimization solving systems of nonlinear equations”. *Computers and Mathematics with Applications* 57 (2009) 1877_1882.

[21] A. Leykin, J.Verschelde, A. Zhao, “Newton's method with deflation for isolated singularities of polynomial systems”, *Theoretical Computer Science*, Volume 359, Issues 1-3, 14 August 2006, Pages 111-122.

[22] W. X. Qian *et al* “He's iteration formulation for solving nonlinear algebraic equations” 2008 *J. Phys.: Conf. Ser.* 96 012192

[23] P. E. Wellstead, “Introduction to Physical System Modeling”. *Control System Principles*, 2000.

[24] J. Kennedy, R. Eberhart, "Particle swarm optimization," *Neural Networks, 1995. Proceedings., IEEE International Conference*, vol.4, no., pp.1942-1948 vol.4, Nov/Dec 1995

[25] J. Lee, “Mixed integer nonlinear programming: some modeling and solution issues”. *IBM Journal of Research and Development.* May-Jul 2007; 51, 3/4; ABI/INFORM Global pg. 489.

[26] R. Stevens, A. Freedman, J. Kreuder, “ThermoElectric Power System Simulator (TEPSS): A Tool for Designing the Next Generation of Thermoelectric Power Technologies,” poster session presented at the International Conference on Thermoelectrics, Munich, Germany, August, 2009.

[27] I. Solberg, “MATLAB fminconset function”. 2000, http://www.mathworks.com/matlabcentral/fileexchange/96 accessed online 10/19/2010.

[28] P. Colonna, T.P. van der Stelt, 2004, *FluidProp: a program for the estimation of thermo physical properties of fluids*, Energy Technology Section, Delft University of Technology, The Netherlands (http://www.FluidProp.com).

[29] F. Wicks, “Thermodynamic analysis of an enhanced gas and steam cycle,” *2002 37th lntersociety Energy Conversion Engineering Conference.* 2002.

[30] A. P. Freedman, “A thermoelectric generation subsystem model for heat recovery simulations” M.S. thesis, Rochester Institute of Technology, Rochester, NY, USA, 2010.

[31] K. D. Smith, “An investigation into the viability of heat sources for thermoelectric power generation systems”, M.S. thesis, Rochester Institute of Technology, Rochester, NY, USA, 2009.

APPENDIX A

User Input and Execution Files

Case Study I – Combined Cycle Optimization

```
%This file is used to execute the optimization process.

format long
clear classes
clear all
clc

%define components
%1-d cell array containing every component in the system
solver_inputs.fstr = '{compressor(obj.parameters.compressor) ,
                    heater(obj.parameters.heater),
                    turbine(obj.parameters.gasturbine),
                    heatx(obj.parameters.heatx),
                    pump(parameters.pump),
                    steamturbine(obj.parameters.steamturbine),
                    condenser(obj.parameters.condenser),
                    pressref(obj.parameters.pressref)}';

%create the nodes by assigning a cell in cell array n to the class
%definition of the node domain.
cpconst = 1004.83; %constant specific heat for gas cycle

solver_inputs.n{1} =
fluidconst('N2,O2,CH4',[.7652,.2035,.0313],'GasMix','PT',cpconst);
solver_inputs.n{2} =
fluidconst('N2,O2,CH4',[.7652,.2035,.0313],'GasMix','PT',cpconst);
solver_inputs.n{3} =
fluidconst('N2,O2,H2O,CO2',[7652,.1410,.0625,.0313],'GasMix','PT',cpconst);
solver_inputs.n{4} =
fluidconst('N2,O2,H2O,CO2',[7652,.1410,.0625,.0313],'GasMix','PT',cpconst);
solver_inputs.n{5} =
fluidconst('N2,O2,H2O,CO2',[7652,.1410,.0625,.0313],'GasMix','PT',cpconst);
solver_inputs.n{6} = fluid('water',1,'IF97','PT');
solver_inputs.n{7} = fluid('water',1,'IF97','PT');
solver_inputs.n{8} = fluid('water',1,'IF97','PT');
solver_inputs.n{9} = fluid('water',1,'IF97','PT');
solver_inputs.n{10} = fluid('water',1,'IF97','PT');
solver_inputs.n{11} = mechrot;
solver_inputs.n{12} = mechrot;
solver_inputs.n{13} = mechrot;
solver_inputs.n{14} = mechrot;


%describe the way that components are connected via nodes. Create a p by q
%array for which p = # of components and q = (1 + the maximum number of
%nodes connected to any one component in the system). Row i in this array must
%correspond to
%the ith component declared in the string solver_inputs.fstr above. Use the
following
```

```
%format for each row [ # of node connections, node #, node#, ... node#].
%If the # of connections is <q-1 for any one component then put a zero as a
%placeholder to fill out the p by q array.
solver_inputs.cnmap =[3,1,-2,11,0;
                      2,2,-3,0,0;
                      4,3,-4,-11,-12;
                      4,4,-5,7,-8;
                      2,6,-7,0,0;
                      4,8,-9,-13,-14;
                      2,9,-10,0,0;
                      2,10,-6,0,0];

%Apply system boundary conditions using an nx3 where n is the number of
%boundary conditions. Use the format [bc value, node #, property #] for
%each row, where property # is defined in the node dommain file.


solver_inputs.bcmap = [1,1,.0001262;%gas side mass flow rate
                       1,2,299.81;  %inlet temperature
                       1,3,101300;  %inlet pressure
                       11,1,.808;   %torque into compressor
                       11,2,60;     %rad/sec into compressor
                       12,1,.6719;  %gas turbine net torque out
                       12,2,60;     %rad/sec out of gas turbine
                       13,1,0.28467;%steam turbine torque out
                       13,2,60;     %steam turbine rad/sec
                       14,1,0;      %unused node torque out
                       14,2,60];    %unused node rad/sec

%Provide an initial guess for the steady state solution of error equations.
%Define an mx3 array for which each row applies to one unknown state. Use
%the format: [guess, node #, Property #] on each line, where the property #
%is defined in the node domain file under the update method.

%number of guesses should equal total number of system states for all nodes
%minus the number of boundary conditions minus (the number of closed loops
%times the number of through variables in that loop)
solver_inputs.xguess = [2,1,.0001;
                        2,2,680;
                        2,3,1418000;
                        3,1,.0001;
                        3,2,1523;
                        3,3,1418000;
                        4,1,.0001;
                        4,2,823;
                        4,3,100000;
                        5,1,.0001;
                        5,2,470;
                        5,3,100000;
                        6,1,.00001429;
                        6,2,310.92;
                        6,3,6551;
                        7,1,.00001;
                        7,2,350;
                        7,3,13000000;
                        8,1,0.00001;
                        8,2,700;
                        8,3,13000000;
                        9,1,0.00001;
                        9,2,325;
                        9,3,7000;
                        10,1,0.00001;
                        10,2,320;
```

```
                        10,3,7050];

solver_inputs.eps = 1e-8; %kickout criteria for state solver. eps = norm of
previous step size

solver_inputs.h = 1e-9; % relative step size for calculating numerical
derivatives. f' =~ (f(x*(1+h)) - f(x/(1+h)))/(2h)
%Theory suggests that h should be <= eps.

%maximization or minimization problem? solver_inputs.minmax = 'min' for
minimization, 'max' for maximization
solver_inputs.minmax = 'min';

%set components as active or passive 0 = active, 1 = passive. Slot number
%corresponds to component number.
solver_inputs.removable = zeros(1,size(solver_inputs.cnmap,1));

%Define all system parameters to an initial value. If the value is not
%being optimized then set it to its final value, otherwise provide an
%initial guess.

%declare all system parameters and set them to a value. For constant
%parameters this will remain the value of the parameter throughout
%simulation. For design variables declared in dvlist below, the specified
%value is an initial guess.

%define cost inputs for use later in cost_function_def
equipment_lifetime = 30; %yrs
%[Ac_elec, Dc_elec, gas, oil, coal, thermal, flow, kinetic, potential]
costperkwh = [0.1,0.1,.0,.18,.04,0,0,0,0];
costperc02 = [0,0,0];

%compressor
parameters.compressor.wc = 49.118; %work in
parameters.compressor.eff = .85;   %compressor efficiency
parameters.compressor.CR = 14;      %compression ratio
parameters.compressor.direction = solver_inputs.cnmap(1,2:4);

parameters.heater.qin = 130;     %heat rate in
parameters.heater.direction = solver_inputs.cnmap(2,2:3);

%gas turbine
parameters.gasturbine.wt = 89.371;%gross power
parameters.gasturbine.eff = .87; %efficiency
parameters.gasturbine.CR = 14; %'decompression' ratio
parameters.gasturbine.direction = solver_inputs.cnmap(3,2:5);
parameters.gasturbine.tmax = 1700; %max temp out

%heat exchanger
parameters.heatx.UA = .5; %overall heat transfer coefficient initial guess
parameters.heatx.flowdir= 'counter';%flow configuration (counter or parallel)
parameters.heatx.pressmax = 1e9;% for penalty function
parameters.heatx.direction = solver_inputs.cnmap(4,2:5);

%pump
parameters.pump.wp = .2315;%power delivered
parameters.pump.eff = .9;%eficiency
parameters.pump.CR = 2103.8487;%pressure ratio
parameters.pump.direction = solver_inputs.cnmap(5,2:4);

%steam turbine
parameters.steamturbine.wt = 17.08;%power output
parameters.steamturbine.eff = .87; %efficiency
```

```
parameters.steamturbine.CR = 2103.8487;%ressure ratio
parameters.steamturbine.direction = solver_inputs.cnmap(6,2:5);

%condenser
parameters.condenser.direction = solver_inputs.cnmap(7,2:5);

%pressure reference component
parameters.pressref.pref = 6551;%reference pressure
parameters.pressref.direction = solver_inputs.cnmap(8,2:3);

%declare design variables
dvlist = {'parameters.heatx.UA','parameters.heater.qin'};

%discrete variables
discrete = {[],[]};

%generate initial guess for design variables
for i =1:length(dvlist)
dvguess(i) = eval(dvlist{i});
end

%update relation for design variables
dvupdate = 'obj.parameters.heatx.UA = obj.dvguess(1); obj.parameters.heater.qin
= obj.dvguess(2);';

%formulate the cost function:
cost_function_def = {'component_cost.cost(2)';             %A cost($)
                     'component_cost.power(1)*8.766*.025'; %B fuel cost per yr
                     'zeros(12,1)';                        %C
                     '0';                                  %D
                     'component_cost.power(2)*8.766';      %E
                     'zeros(12,1)';                        %F
                     zeros(1,12);                        %Cost per unit for C
                     zeros(1,12);                        %Cost per unit for F
                     equipment_lifetime};%Time by which to multiply B,C,E and F

%create optimization shell
C = optimsolve(parameters,dvguess, solver_inputs,dvupdate,cost_function_def);

%determine if the correct number of BCs and xguesses are supplied.
C.statecheck

%set convergence criteria
fmincon_options = optimset('UseParallel','always','Tolx', 1e-8,'TolFun',1e-10,
'MaxFunEvals', 250);

%set upper and lower constraints on each DV in the order that they appear
%in dvlist
lb=[.1,100];%lower bounds
ub= [2.8,140]; %upper bounds

%run optimization
[optimalx,net_power,exitflag,output,lambda,grad,hessian] =
C.optimize(fmincon_options,ub,lb,discrete);
```

Case Study II – Optimization of Simple Brayton Cycle with Thermoelectric Heat Recovery

```
%This file is used to execute the optimization process.

format long
clear classes
clear all
clc

%create components in a 1-d cell array containing every component in the system
solver_inputs.fstr = '{compressor(parameters.compressor) ,
                       tepowerunit7(parameters.tepowerunit),
                       heater(parameters.heater),
                       turbine(parameters.turbine),
                       generator(parameters.generator),
                       rotationalreference(parameters.rotationalreference)}';

%create the nodes by assigning a cell in cell array solver_inputs.n to the class
%definition of the node domain.
for i=1:6
    solver_inputs.n{i} = fluid('N2,O2,CH4',[.75,.195,.055],'GasMix','PT');
end
solver_inputs.n{7} = mechrot;
solver_inputs.n{8} = mechrot;
solver_inputs.n{9} = mechrot;
solver_inputs.n{10} = electrical;

%describe the way that components are connected via nodes. Create a p by q
%array in which p = # of components and q = (1 + the maximum number of
%nodes connected to one component). Row i in this array must correspond to
%the ith component declared in the string solver_inputs. fstr above. Use the
following
%format for each row [ # of node connections, node #, node#, ... node#].
%If the # of connections is < q-1 for any one component then put a zero as a
%placeholder to fill out the p by q array.
solver_inputs.cnmap =[3,1,-2,7,0;
                      4,2,3,5,6;
                      2,3,-4,0,0;
                      4,4,-5,7,8;
                      3,8,-9,-10,0;
                      1,9,0,0,0];

%Apply system boundary conditions using an nx3 where n is the number of
%boundary conditions. Use the format [bc value, node #, property #] for
%each row, where property # is defined in the node dommain file.

solver_inputs.bcmap = [6,3,101300;  % gas cycle outlet pressure = 1 atmosphere
                       1,2,300;     % gas cycle inlet temperature = 300 K
                       1,3,101300;  % gas cycle inlet pressure = 1 atmosphere
                       9,2,0];      % reference rotational velocity is zero

%Provide an initial guess for the steady state solution of component equations.
%Define an mx3 array for which each row applies a guess to one unknown. Use
%the format: [node #, Property #, guess] on each line, where the property #
```

```
%is defined in the node domain file in the update method.
solver_inputs.xguess = [1,1,30;
                        2,1,30;
                        2,2,500;
                        2,3,3.5e6;
                        3,1,30;
                        3,2,650;
                        3,3,3.4e6;
                        4,1,30;
                        4,2,1700;
                        4,3,3.3e6;
                        5,1,30;
                        5,2,1000;
                        5,3,2e5;
                        6,1,30;
                        6,2,900;
                        7,1,1e6;
                        7,2,60;
                        8,1,4e6;
                        8,2,60;
                        9,1,0;
                        10,1,70;
                        10,2,3e6];

solver_inputs.eps = 1e-5; %kickout criteria for solver convergence

solver_inputs.h = 1e-6; % relative step size for calculating numerical derivatives.
f' = (f(x*(1+h)) - f(x/(1+h)))/(2h)
%Theory suggests that h should be < eps.

%maximization or minimization problem? solver_inputs.minmax = 'min' for
%minimization, 'max' for maximization
solver_inputs.minmax = 'min';

%set components as active or passive 0 = active, 1 = passive. Slot number
%corresponds to component number. length(solver_inputs.removable) MUST = #
%of compoinents in the system.
solver_inputs.removable = [0,0,0,0,0,0];


%declare all system parameters and set them to a value. For constant
%parameters this will remain the value of the parameter throughout
%simulation. For design variables declared later in dvlist, the specified
%value is an initial guess.

%set cost constants for use in cost_function_def below
equipment_lifetime = 30; %yrs
%[Ac_elec, Dc_elec, gas, oil, coal, thermal, flow, kinetic, potential]
costperkwh = [0.1,0.1,.0,.18,.04,0,0,0,0];
%[CO2, NOx, SOx]
costperc02 = [0,0,0];

parameters.compressor.CR = 35; %compression ratio
parameters.compressor.wc=6400000; %power delivered to fluid
parameters.compressor.shaftspeed = 60; %rad/sec shaft input
parameters.compressor.direction = solver_inputs.cnmap(1,2:4); %expected inlets and
outlets
```

```
%Hot side fins (Rectangular)
parameters.tepowerunit.fins.t_h=1e-3;        %fin thickness
parameters.tepowerunit.fins.l_h=.01;         %fin length
parameters.tepowerunit.fins.num_h = 7241.4;  %# of fins
parameters.tepowerunit.fins.k_h=250;         %fin conductivity
parameters.tepowerunit.fins.base_t_h=.0075;  %fin array base thickness

%Cold side fins (Rectangular)
parameters.tepowerunit.fins.t_c=1e-3;
parameters.tepowerunit.fins.l_c=.01;
parameters.tepowerunit.fins.num_c=7241.4;
parameters.tepowerunit.fins.k_c=250;
parameters.tepowerunit.fins.base_t_c=.0075;

parameters.tepowerunit.unit.series=10;      %zone length in modules
parameters.tepowerunit.unit.parallel=1200;  %zone width in modules
parameters.tepowerunit.unit.num=20;         % # of zones

%Insulation Between Base Plates
parameters.tepowerunit.unit.insul_k=.05;                %insulation conductivity
parameters.tepowerunit.unit.zone_to_mod_area_ratio=1.01;% area ratio of zone to
total module area

%Thermal Contact Resistance
parameters.tepowerunit.unit.therm_contact_res=.00001;   %zone to module contact
resistance

%Environmental Losses
parameters.tepowerunit.unit.uvalue=.01;             %heat transfer coefficient
between power unit and environment
parameters.tepowerunit.unit.shell_t=1;              %insulation thickness
parameters.tepowerunit.unit.shell_k=.000001;        %Conductivity to environment
parameters.tepowerunit.unit.envir_temp=300;         %environment temperature

%Needed for Option2 - thermoelectric properties
%p Leg
parameters.tepowerunit.module.rho_p=14e-6; %ohm*m
parameters.tepowerunit.module.alpha_p=4e-4;%total alpha
parameters.tepowerunit.module.k_p=1;%w/(m*K)
parameters.tepowerunit.module.l_p=.005; %leg length
parameters.tepowerunit.module.area_p=(1.397e-3)^2;% for 1 leg

%n Leg
parameters.tepowerunit.module.rho_n=14e-6;
parameters.tepowerunit.module.alpha_n=0e-4;%(all seebeck is on p leg parameters)
parameters.tepowerunit.module.k_n=1;
parameters.tepowerunit.module.l_n=.005;
parameters.tepowerunit.module.area_n=(1.397e-3)^2;

parameters.tepowerunit.module.a_ratio=2.7; %area ratio of TE material to ceramic
parameters.tepowerunit.module.num=127;%number of pairs

%Ceramic
parameters.tepowerunit.module.l_cer=.8e-3;%ceramic thickness
parameters.tepowerunit.module.k_cer=25;%ceramic conductivity
```

```
%Electrical Contact Resistance
parameters.tepowerunit.module.contact_resist=3e-9;  %This is a guess, but is in the
middle of range
                                                    %of ones reported in the
                                                    %literature. - electrical contact
                                                    %resistance



%Module 'option1','option3','option4','option5' Costs
parameters.tepowerunit.cost.specific.module=2.5; %$/module
%Module 'option2' Costs
parameters.tepowerunit.cost.specific.module_p_leg_material=8e7; %$/m3
parameters.tepowerunit.cost.specific.module_n_leg_material=8e7; %$/m3
parameters.tepowerunit.cost.specific.module_ceramic=0; %$/m3
parameters.tepowerunit.cost.specific.cost_per_leg_pair=0; %$/leg pair
parameters.tepowerunit.cost.fixed.module_manufac=0; %$
%Fin Costs
parameters.tepowerunit.cost.specific.fin_material=4e4; %$/m3
parameters.tepowerunit.cost.fixed.fin_manufac_h=0; %$
parameters.tepowerunit.cost.fixed.fin_manufac_c=0; %$
%Insulation Costs
parameters.tepowerunit.cost.specific.insulation=0; %$/m3
%Cost Per Zone Area
parameters.tepowerunit.cost.specific.cost_per_zone_area=0; %$/zone area
%Other Costs
parameters.tepowerunit.cost.fixed.other=0; %$
%Fixed Assembly Costs
parameters.tepowerunit.cost.fixed.assembly=1; %$

parameters.tepowerunit.direction = solver_inputs.cnmap(2,2:5); %inlets and outlets

%COUNTER FLOW, declare fin configuration and thermoelectric parameters
%defined
parameters.tepowerunit.options={'option2','','straightfins_aligned','straightfins_a
ligned'};

                          %This is used to tell the object how to calculate
                          %the module parameters from geometrical and
                          %material properties.

parameters.heater.toutmax = 1700; %max temp out
parameters.heater.LHV = 50000000;%lower heating value
parameters.heater.stoichafr = 17.2;%ideal air fuel ratio
parameters.heater.direction = solver_inputs.cnmap(3,2:3);


parameters.turbine.tmax = 1700; %max temp in - for penalty function
parameters.turbine.efficiency = 0.85;
parameters.turbine.shaftspeed = 60; %rad/s shaft output speed
parameters.turbine.direction = solver_inputs.cnmap(4,2:5);

parameters.generator.damperresistance = 5555;%generator damping
parameters.generator.Kv = 5555;%voltage constant
parameters.generator.direction = solver_inputs.cnmap(5,2:4);
```

```
parameters.rotationalreference.refspeed = 0;%rotational speed of shaft reference
(rad/s)
parameters.rotationalreference.direction = solver_inputs.cnmap(6,2);

%declare design variables
dvlist =
{'parameters.tepowerunit.fins.num_h','parameters.tepowerunit.unit.series'};
discrete = {[],[],};
for i =1:length(dvlist) %get IC's for DVs from parameters structure
dvguess(i) = eval(dvlist{i});
end

%set hot and cold side fins.num = to design variable guess 1 at each
%iteration, do the same for modules in thermal series and guess 2 (single line of
code)
dvupdate = 'parameters.tepowerunit.fins.num_h = obj.dvguess(1);
parameters.tepowerunit.fins.num_c = obj.dvguess(1);
parameters.tepowerunit.unit.series = obj.dvguess(2)';%update relation for design
variables

%formulate the cost function:
cost_function_def = {'component_cost.cost(1)';          %A fixed cost($)
                     'component_cost.power(3)*0.025*8.766'; %B fuel cost($/yr)
                     '0';                               %C
                     '0';                               %D
                     '(component_cost.power(2))*8.766';%E KWh/yr
                     '0';                               %F
                     zeros(1,12);                       %Cost per unit for C
                     zeros(1,12);                       %Cost per unit for F
                     equipment_lifetime};   %Time by which to multiply B,C,E and F


%create optimization shell
C = optimsolve(parameters,dvguess, solver_inputs,dvupdate,cost_function_def);

%determine if the correct number of BCs and xguesses are supplied.
C.statecheck

%set convergence criteria
fmincon_options = optimset('UseParallel','always','Tolx', 1e-6, 'TolFun',1e-
8);%,'DiffMinChange',1e-4);

%set upper and lower constraints on each DV in the order that they appear
%in dvlist

lb=[6000,2];%lower bounds
ub= [10500,15]; %upper bounds

%run optimization
[optimalx,net_power,exitflag,output,lambda,grad,hessian] =
C.optimize(fmincon_options,ub,lb,discrete);
```

APPENDIX B

Shell Class Definitions

Optimization Shell (optimsolve.m)

```
classdef optimsolve < handle
    properties
        parameters
        A
        ss_soln
        dvguess
        options
        solver_inputs
        dvupdate
        cfdef
        firstcost
    end

    methods

        function obj = optimsolve(parameters,dvguess,solver_inputs,
                 dvupdate, cost_function_def)%accept dv's from
                  run_optimization.m

        %store inputs as properties
        obj.parameters = parameters;
        obj.solver_inputs = solver_inputs;
        obj.dvupdate = dvupdate;
        obj.dvguess = dvguess;
        obj.cfdef = cost_function_def;

        %distribute boundary conditions and initial guesses to nodes.
        %if node is 'fluid' then look up specific enthalpy along the
            way.
        for ii = 1: size(solver_inputs.bcmap,1)
            if
            strcmp(class(solver_inputs.n{solver_inputs.bcmap(ii,1)}),
             'fluid')==1 ||
            strcmp(class(solver_inputs.n{solver_inputs.bcmap(ii,1)}),
             'fluidconst')==1
            solver_inputs.n{solver_inputs.bcmap(ii,1)}
            .initial_update(solver_inputs.bcmap(ii,:));
            else
            solver_inputs.n{solver_inputs.bcmap(ii,1)}
            .update(solver_inputs.bcmap(ii,:));
            end
        end

 %do the same for initial guesses
        for ii = 1: size(solver_inputs.xguess,1)
```

```
            if
            strcmp(class(solver_inputs.n{solver_inputs.xguess(ii,1)}),
            'fluid')==1||
            strcmp(class(solver_inputs.n{solver_inputs.xguess(ii,1)}),
            'fluidconst')==1
            solver_inputs.n{solver_inputs.xguess(ii,1)}
            .initial_update(solver_inputs.xguess(ii,:));
            else
            solver_inputs.n{solver_inputs.xguess(ii,1)}.
            update(solver_inputs.xguess(ii,:));
            end
        end

%if node is fluid, lookup enthalpy and other properties
        for ii = 1:length(solver_inputs.n)
             if
            strcmp(class(solver_inputs.n{ii}),'fluid')==1||
            strcmp(class(solver_inputs.n{ii}),'fluidconst')==1
            solver_inputs.n{ii}.lookup;
            else
            end
        end

%replace fluid T/q bcs with enthalpy lookup if node is fluid
        for ii = 1:size(solver_inputs.bcmap,1)
            if
            strcmp(class(solver_inputs.n{solver_inputs.bcmap(ii,1)}),
            'fluid')==1||
            strcmp(class(solver_inputs.n{solver_inputs.bcmap(ii,1)}),
            'fluidconst')==1
            if solver_inputs.bcmap(ii,2) == 2
                solver_inputs.bcmap(ii,3) =
                  solver_inputs.n{solver_inputs.bcmap(ii,1)}.enthalpy;
            else
            end
            end
        end

%replace fluid T/q guesses with enthalpy lookup if node is fluid
        for ii = 1:size(solver_inputs.xguess,1 )
           if
            strcmp(class(solver_inputs.n{solver_inputs.xguess(ii,1)}),
            'fluid')==1||
            strcmp(class(solver_inputs.n{solver_inputs.xguess(ii,1)}),
            'fluidconst')==1
            if solver_inputs.xguess(ii,2) == 2
                solver_inputs.xguess(ii,3) =
                  solver_inputs.n{solver_inputs.xguess(ii,1)}.enthalpy;
            else
            end
            end
        end
   %replace user supplied guesses and bcs with ones containing enthalpy
        obj.solver_inputs.xguess = solver_inputs.xguess;
        obj.solver_inputs.bcmap = solver_inputs.bcmap;
        end %end constructor
```

```
 %evaluate cost function
   function cost = objective_f(obj,fmincon_dvguess)

 %grab parameters from properties
   parameters = obj.parameters;

 %store guess as property
   obj.dvguess = fmincon_dvguess;

 %update parameters structure with new design variables
   eval(obj.dvupdate)

   %grab cost function constants from properties
   cfdef = obj.cfdef;

%import previous steady state solution as the new initial guess if
%one exists.
   if isempty(obj.ss_soln) == 0
       obj.solver_inputs.xguess(:,3) = obj.ss_soln;
   else
   end

   %create components from user supplied string
   obj.solver_inputs.f = eval(obj.solver_inputs.fstr);

   %create simulator shell
   obj.A = newtonsolve2(obj.solver_inputs);

   %check feasibility
   y = obj.A.simulation_feasible;

   %skip simulation if sum(y)>0
   if y>0
       if isempty(obj.firstcost)==1
           disp('Infeasible starting point resulting in unsolvable
                   simulation. Refine starting point for
                   and component parameters')
           return
       else

            disp('Infeasible design variables - likely to result
                    in unsolvable simulation, consider changing
                   upper or lower bounds of DVs')
            disp('attempting to penalize cost function in
                   infeasible space')
            cost = abs(obj.firstcost)*10^4;
            disp('Designs Varible Values:')
            disp(obj.dvguess)
            disp('cost')
            disp(cost)
       return
       end

   %if system is feasible run simulation and get costs
```

```
elseif y==0
    obj.ss_soln = obj.A.iterate;
    all_component_costs = obj.A.cost;
else
end

phys0 = 0;

%calculate penalty function
for i = 1:length(all_component_costs)
     component_cost = all_component_costs(i);
    phys0 = phys0+component_cost.physcon;
for j = 1:6
cost_func{i,j} = eval(cfdef{j}); %evaluate user defined cost
                                    function inputs
end
end

%sum like costs
    A0=0;
    B0=0;
    C0=0;
    D0=0;
    E0=0;
    F0=0;

    costperunitC = cfdef{7};
    costperunitF = cfdef{8};
    time = cfdef{9};
  for i = 1:length(all_component_costs)
      A0 = A0+cost_func{i,1};
      B0 = B0+cost_func{i,2}*time;
      C0 = C0+cost_func{i,3}'*costperunitC'*time*8.766;
      D0 = D0+cost_func{i,4};
      E0 = E0+cost_func{i,5}*time;
      F0 = F0+cost_func{i,6}'*costperunitF'*time*8.766;
  end


  %evaluate cost fn
%if maximizing the cost function, negate the value

if strcmp(obj.solver_inputs.minmax,'min') ==1
    cost = ((A0+B0+C0)/(D0+E0+F0))*(1+phys0*1e-2); %calculate
                                                    cost

elseif strcmp(obj.solver_inputs.minmax,'max')==1
    cost = -((A0+B0+C0)/(D0+E0+F0))*(1-phys0*1e-2); %calculate
                                                    cost
else
    disp('solver_inputs.minmax must be set to either min or
          max')
end

%display the steady state solution for the system and current
```

```
  %design variable values. If there are fluid nodes, look up the
  %temperature that corresponds to the enthalpy solution and
       report
  %the temperature in place of the enthalpy.
 ss_soln2 = obj.ss_soln;
 for ii = 1:size(obj.solver_inputs.xguess,1)%replace fluid T/q
      guesses with enthalpy lookup if node is fluid
      if
strcmp(class(obj.solver_inputs.n{obj.solver_inputs.xguess(ii,1)},
'fluid')==1 ||
strcmp(class(obj.solver_inputs.n{obj.solver_inputs.xguess(ii,1)},
'fluidconst')==1
      if obj.solver_inputs.xguess(ii,2) == 2
      ss_soln2(ii) =
      obj.solver_inputs.n{obj.solver_inputs.xguess(ii,1)}.temp;
      else
      end
      end
  end


  disp('ss_soln')
  disp(ss_soln2)


  disp('dv values')
  disp(obj.dvguess)

  disp('objective function value')
  if strcmp(obj.solver_inputs.minmax,'min') ==1
      disp(cost)
  elseif strcmp(obj.solver_inputs.minmax,'max')==1
      disp(-cost)
  end

  end %end cost function evaluation

  %call fmincon
function [optimalx,net_power,exitflag,output,lambda,grad,hessian]
            = optimize(obj,options,ub,lb,discrete)

      %display a message if the initial guess is outside of the
      %design space for a variable.
      for ii = 1:length(obj.dvguess)
          if obj.dvguess(ii)<lb(ii)
              disp('initial guess for a design variable is lower
                   than the lower boundary of the design space.
                   design variable # is')
              disp(ii)
              disp('the value of the initial guess is being
                  adjusted to match the lower bound specified')
              obj.dvguess(ii) = lb(ii);
          else
          end
          if obj.dvguess(ii)>ub(ii)
```

```
                    disp('initial guess for a design variable is higher
                        than the upper boundary of the design space.
                        design variable # is')
                    disp(ii)
                    disp('the value of the initial guess is being
                        adjusted to match the upper bound specified')
                    obj.dvguess(ii) = ub(ii);
                else
                end
            end
        %run the simulation once at the initial point, store this cost
            obj.firstcost = obj.objective_f(obj.dvguess);

            %count discrete variables
            for r = 1:length(discrete)
            g(r) = sum(discrete{r}.^2);
            end

            %if there are no discrete variables, run fmincon
            g = sum(g);

            if g==0 %use fmincon for all continuous variables,
                    otherwise use fminconset
            [optimalx,net_power,exitflag,output,lambda,grad,hessian]  =
fmincon(@obj.objective_f,[obj.dvguess],[],[],[],[],lb,ub,[],options);
            else

            %If there are discrete variables run fminconset
            [optimalx,net_power,exitflag,output,lambda,grad,hessian]  =
fminconset(@obj.objective_f,[obj.dvguess],[],[],[],[],lb,ub,[],options,
discrete,[]);
            end
        end %end optimize function

        %count node variables
        function statecount = checkninputs(obj,n)%read the number of
             states from each node by calling the numstates method in
             each node
            for i = 1:length(n)
                z(i) = n{i}.numstates;
            end
            statecount = sum(z);
        end

   function statecheck(obj) %verify the correct # of xguesses and BCs
                               are provided

        numstates = obj.checkninputs(obj.solver_inputs.n);
        numguess = size(obj.solver_inputs.bcmap,1)+
                        size(obj.solver_inputs.xguess,1);

     %is the number of bcs and guesses provided equal to the number of
        %node variables? If yes, proceed, if no, display warnings.
        if numstates - numguess(1) ==0
```

```
        elseif numstates - numguess<0
            disp('WARNING: combined # of BCs and state guesses (xguess
                  and bcmap) is too large. There should be a combined
                  number of')
            disp(numstates)
            disp('rows in the two arrays. Some BCs may not be
                  satisfied')
            disp(numguess(1))
            disp('values are provided')
            r = input('enter 1 to continue, 0 to break');
            if r ==0
            return
            elseif r==1
            else
                disp('invalid entry, stopping routine...')
            return
            end

        else
            disp('WARNING: # of combined BCs and state guesses (xguess
                   and bcmap) is too small. There should be a combined
                  number of')
            disp(numstates)
            disp('rows in the two arrays')
            disp(numguess(1))
            disp('values are provided')
            r = input('enter 1 to continue, 0 to break');
            if r ==0
            return
            elseif r==1
            else
                disp('invalid entry, stopping routine...')
                return
            end
        end


    end
    end

end
```

Simulator shell (newtonsolve2.m)

```
classdef newtonsolve2<handle
%Solve a coupled system of nonlinear algabraic equations defined in a
%series of components. The method Leverages Newton's Method in n
%dimensions and uses the centered difference  method to approximate the
%Jacobian matrix at a given point.


    properties
        f
        x
        J
```

```
        fval
        deltax
        h
        eps
        n
        cnmap
        xguess
        nodes
        bcmap
        input
        onoff
        Jinv
    end

    methods
        function obj = newtonsolve2(solver_inputs) %constructor

            %store inputs as properties
            obj.f = solver_inputs.f;
            obj.bcmap = solver_inputs.bcmap;
            obj.x = solver_inputs.xguess;
            obj.eps = solver_inputs.eps;
            obj.h = solver_inputs.h;
            obj.cnmap = solver_inputs.cnmap;
            obj.n = solver_inputs.n;
            obj.xguess = solver_inputs.xguess(:,3);
            obj.onoff = solver_inputs.removable;


            %remove zeros from cnmap to produce a variable 'nodes'
            for ii = 1:size(obj.cnmap,1)
            for jj = 2:size(obj.cnmap,2)
                if obj.cnmap(ii,jj)==0
                   obj.nodes{ii,jj-1} = []; %if cnmap contains a zero,
                                            leave the cell empty
                else
                   obj.nodes{ii,jj-1} =
solver_inputs.n{abs(obj.cnmap(ii,jj))}; %otherwise put the appropriate
                                           node into that cell
                end
            end
            end

            numeq=0;

            %count the equations in the system
            for ii = 1:size(obj.cnmap,1)
            for jj = 1:obj.cnmap(ii,1)
                input{1,jj} = obj.nodes{ii,jj};
            end
            numeq(ii) = length(solver_inputs.f{ii}.compute(input{1,:},
                        obj.onoff(ii)));

        %produce cell array of nodes to feed to each component, store
        %for later.
```

```
    obj.input{ii} = input;
    clear input
    end

    %compare the number of equations to the number of guesses
    %display a warning if the variables aren't equal.
    %check for a square system (# of x guesses == number of
          equations)
    %display an error if the values are unequal.
    if length (obj.x(:,1)) == sum(numeq)
    else
        disp('length of x guess vector must == number of
          equations'); %display error if system is non-square
        disp('number of equations ');
        disp(sum(numeq));
        disp('length of x guess vector');
        disp(length(obj.xguess));
    end
end %end constructor

function J = jacobian(obj)
    %find jacobian using centered difference method

    %import properties for calculation
    x=obj.x;
    n=obj.n;


    for i = 1:length(obj.f)%for each component
            input = obj.input{i};
            for j = 1:size(x,1)%for each unknown in the system

       % use relative step size for numerical derivative
       % as long as the value of the guess is not close to zero
                if abs(x(j,3))>10*obj.eps
                    %step towards zero, update x and calculate
                    %error for each component
                    x(j,3) = x(j,3)/(1+obj.h);
                    w = x(j,1);
                    n{w}.update(x(j,:));
                    hminus = obj.f{i}.compute(input{1,:},
                       obj.onoff(i));

                    %step away from zero, repeat calculation
                    x(j,3) = x(j,3)*(1+obj.h)^2;
                    n{w}.update(x(j,:));
                    hplus=obj.f{i}.compute(input{1,:},
                      obj.onoff(i));

                %use centered difference method to calculate a
                %partial derivative of equation k in component
                %i with respect to node variable j.
                 for k = 1:length(obj.f{i}.compute(input{1,:},
                             obj.onoff(i)))
                    if abs(x(j,3)) >10*obj.eps
```

```
                    Jcomp(k,j) = (hplus(k)-
                    hminus(k))/(x(j,3)-x(j,3)/(1+obj.h)^2);
                    %return x to original value
                    x(j,3) = x(j,3)/(1+obj.h);
                  end
                end

                elseif abs(x(j,3))<=10*obj.eps
              %use fixed step size (h) to calculate the partial
               %derivative
                x(j,3) = x(j,3)+obj.h;
                w = x(j,1);
                n{w}.update(x(j,:));
                n{w}.lookup;
                hminus = obj.f{i}.compute(input{1,:},
                      obj.onoff(i));
                x(j,3) = x(j,3)-2*obj.h;%step away from zero,
                                        repeat calcualtion
                n{w}.update(x(j,:));
                n{w}.lookup;
                hplus=obj.f{i}.compute(input{1,:},
                      obj.onoff(i));

                for k = 1:length(obj.f{i}.compute(input{1,:},
                       obj.onoff(i)))
                    if abs(x(j,3)) >0
                        Jcomp(k,j) = (hplus(k)-
                                      hminus(k))/(2*obj.h);
                        %return x to original value
                        x(j,3) = x(j,3)+obj.h;
                    else
                    end


                end



                end
              end
q{i} = Jcomp;% get a cell array of each component's Jacobian
clear Jcomp
 end


 for i=1:length(obj.f)
      input = obj.input{i};
      s{i} = obj.f{i}.compute(input{1,:}, obj.onoff(i))';
      %split the 3-d array of component jacobians into a
       %cell array of 2-d arrays, each cell containing J for
       %a component
 end


  r = q{1};%prepare to concatenate
  obj.fval = s{1};

  if length(obj.f)==1
      obj.J = r;
```

```
        else
            for j=2:length(s)
                obj.fval = [obj.fval;s{j}];%concatenate residuals
                r = [r;q{j}];%concatenate jacobians of each
                                component into a single array
            end
            obj.J = r;
        end

        t = size(obj.J);
       if t(1)==t(2)
        J = obj.J;

    %perform solvability checks:
    %make sure J is square, if not, display a warning
    %also make sure there are no zero rows or columns and that the
    %determinant is non-zero.
        for vv = 1:size(J,1)
        rownorm(vv) = sum(J(vv,:).^2);
        colnorm(vv) = sum(J(:,vv).^2);
        if rownorm(vv) == 0
            disp('WARNING: Jacobian has a zero row and is
                    therefore singular. ROW #:')
            disp(vv)
            disp('The corresponding equation does not depend on
                    any node variables')
            disp(J)
        else
        end
        if colnorm(vv) == 0
            disp('WARNING: Jacobian has a zero column and is
                   therefore singular. COL #:')
            disp(vv)
            disp('no equations depend on the corresponding node
                    variable')
            disp(J)
        else
        end
        end
       else disp('error - non square jacobian, not enough
                  variables or equations')
           J=obj.J;
           disp(J);

       end
       if det(J) == 0
           disp('Jacobian matrix has determinant of 0. One or more
                    system equations may be dependent on others')
       else
       end
end

        %check components for infeasible inputs
       function y = simulation_feasible(obj)
           for i = 1:length(obj.f);
               y(i) = obj.f{i}.paramcheck;
```

```
        end
        y=sum(y.^2);
    end



    function xstar = iterate(obj)
    % Run the Jacobian method iteratively, using its inverse to
    % calculate a step size in between iterations according to
    % Newton's Method.

        obj.deltax = 1;
        %iterate until step size converges to zero

        %run until step size approaches zero
        while sum(abs(obj.deltax./obj.xguess)) > obj.eps
            J = obj.jacobian;
            Jinv = inv(J);
            %calculate step size (Newton's method)
            obj.deltax = real(Jinv*((-1)*obj.fval));
            %update guess point
            obj.xguess = obj.xguess+ obj.deltax;
            obj.x(:,3)=obj.xguess;
            xstar = obj.xguess;
            obj.Jinv = inv(obj.J);
            obj.Jinv = Jinv;
            %select outputs to display at each iteration
            %disp('delta-x')
            %disp(obj.deltax)
            %disp('xguess')
            %disp(obj.xguess)
            disp('residuals')
            disp(obj.fval)


        end

%iterate until residuals approach zero if they haven't already.
%give up after 100 iterations
        if sum(abs(obj.fval.^2))>obj.eps
            i=1;
            while sum(sqrt(obj.fval.^2))>obj.eps && i<100
            J = obj.jacobian;
            Jinv = inv(J);
            obj.deltax = real(Jinv*((-1)*obj.fval));
            obj.xguess = obj.xguess+ obj.deltax;
            obj.x(:,3)=obj.xguess;
            xstar = obj.xguess;
            obj.Jinv = inv(obj.J);
            obj.Jinv = Jinv;
            i=i+1;
            %disp('delta-x')
            %disp(obj.deltax)
            %disp('xguess')
            %disp(obj.xguess)
```

```
                    disp('residuals')
                    disp(obj.fval)
                    end
                    if sum(abs(obj.fval.^2))<obj.eps
                    else
                        disp('no solution found: 1) a solution to the
                              component equations does not exist, or 2)
                               the solver is stuck at a local min,
                              adjust the initial guess')
                    end
                else
                end
            end


        function munny = cost(obj)%compute cost for each component, sum
                                  %to find total system cost.
                for i = 1:length(obj.f)
                   d(i,:) = obj.f{i}.cost';
                end
                munny = d;
            end
        end

end
```

APPENDIX C

Component and Domain Class Definitions

I. Components

Component: Compressor (compressor.m)

```
classdef compressor<handle


    properties
     eta
     parameters
     onoff
     wc
    end

    methods

        function obj = compressor(parameters)
            obj.parameters = parameters;
        end

        function e=compute(obj, node1, node2,node3,onoff)

%rename nodes
        fluidin = node1;
        fluidout = node2;
        shaftout = node3;
        obj.onoff = onoff;

%rename node variables
        mdotin = fluidin.mdot;
        tempin = fluidin.temp;
        pressin = fluidin.press;
        hin = fluidin.enthalpy;

        mdotout = fluidout.mdot;
        tempout = fluidout.temp;
        pressout = fluidout.press;
        hout = fluidout.enthalpy;

        torque = shaftout.torque;
        angvel = shaftout.angvel;

        eta = obj.parameters.efficiency;
        obj.wc = obj.parameters.wc;

        if onoff ==0 % if component is active
%engineering model equations
 e(1) = mdotin*sign(obj.parameters.direction(1))+
```

```
            mdotout*sign(obj.parameters.direction(2));
 e(2) = mdotin*sign(obj.parameters.direction(1))*
            (hout-hin)-torque*angvel;
 e(3) = obj.parameters.CR*pressin - pressout;

        else %if component is passive
            e(1) = tempin-tempout;
            e(2) = pressin - pressout;
            e(3) = mdotin-mdotout;
        end

        end

        function component_cost = cost(obj)%compute cost of operating
                                            the component
                                           %under steady state
                                           conditions.
         if obj.onoff == 0
           component_cost.cost = [100;0];
           component_cost.power = [0,-obj.wc, 0,0,0,
                  obj.wc*obj.eta-obj.wc,-obj.wc*obj.eta,0,0]';
           component_cost.emissions = [0;0;0];
           component_cost.physcon = 0;
            else
                component_cost.cost = [0;0;0];
                component_cost.power = [0,0,0,0,0,0,0,0,0]';
                component_cost.emissions = [0,0,0]';
                component_cost.physcon = 0;
            end

        end
        function y = paramcheck(obj)
            if obj.parameters.CR>30
                y=1;
            else
            y=0;
            end
        end
    end

end
```

Similar components (not shown) included in the basic component package of TEPSS are Turbine (turbine.m), and Pump (pump.m).

Component: Heater (heater.m)

```
classdef heater<handle

    properties
       qin
```

```
   parameters
   onoff
   fluidprop
end

methods

    function obj = heater(parameters)
        obj.parameters = parameters;
    end

    function e=compute(obj, node1, node2,onoff)

    %rename nodes
    fluidin = node1;
    fluidout = node2;

    obj.onoff = onoff;

    %rename node variables
    mdotin = fluidin.mdot;
    tempin = fluidin.temp;
    pressin = fluidin.press;
    hin = fluidin.enthalpy;

    mdotout = fluidout.mdot;
    tempout = fluidout.temp;
    pressout = fluidout.press;
    hout = fluidout.enthalpy;

        if onoff == 0 %if component is active

         %engineering model
         e(1) = mdotin*sign(obj.parameters.direction(1))*(hout –
                    hin) - obj.parameters.qin; %energy change
         e(2) = pressin - pressout - 50*mdotin^2; %pressure drop
         e(3) = mdotout*sign(obj.parameters.direction(2))+
                    mdotin*sign(obj.parameters.direction(1));%cons.
                                                          of mass

        else %if component is passive
            e(1) = tempin - tempout;
            e(2) = pressin - pressout;
            e(3) = mdotin-mdotout;
        end

    end

    function component_cost = cost(obj)
        %compute cost of operating the component
        %for the state values given by the nodes.
        if obj.onoff ==0
        component_cost.cost = [0;0];
        component_cost.power = [obj.parameters.qin,0,0,0,0,-
```

```
                                obj.parameters.qin,0,0,0]';
            component_cost.emissions = [0;0;0];
            component_cost.physcon = 0;
            else
            component_cost.cost = [0;0];
            component_cost.power = [0,0,0,0,0,0,0,0,0,0]';
            component_cost.emissions = [0;0;0];
            component_cost.physcon = 0;
            end

        end

        function y = paramcheck(obj)
            y=0;
        end
    end
end
```

A similar component (not shown) is the Condenser component (condenser.m).

Component: Heat Exchanger (heatx.m)

```
classdef heatx<handle
    properties
        parameters
        onoff
    end

    methods
        function obj = heatx(parameters)
        obj.parameters=parameters;%store parameters for use by other
                                  methods
        end

        function e = compute(obj, node1, node2, node3, node4, onoff)

            %name the nodes so that equations are easy to read
            hfluidin=node1; %hot side inlet node
            hfluidout=node2;%hot outlet node
            cfluidin=node3; %cold inlet node
            cfluidout=node4;%cold outlet node
            obj.onoff = onoff;

            %Calculate log mean temperature difference
            if strcmp(obj.parameters.flowdir, 'parallel')==1
                        dtln=(((hfluidin.temp - cfluidin.temp) -...
                         (hfluidout.temp - cfluidout.temp))...
                          /log(abs(hfluidin.tempcfluidin.temp)/...
                          abs(hfluidout.temp-cfluidout.temp)));
            elseif strcmp(obj.parameters.flowdir, 'counter')==1
                       dtln=(((hfluidin.temp - cfluidout.temp) -...
                       (hfluidout.temp - cfluidin.temp))...
                        /log(abs(hfluidin.temp-cfluidout.temp)/...
                         abs(hfluidout.temp-cfluidin.temp)));
```

```
            else
                disp('heat exchanger parameters.flowdir must be set to
                        either parallel or counter')
                return
            end

            Qhx = obj.parameters.UA*dtln;


            e(1) = cfluidin.mdot*sign(obj.parameters.direction(1))+...
                cfluidout.mdot*sign(obj.parameters.direction(2));

            e(2) = cfluidin.mdot*sign(obj.parameters.direction(1))*...
                (cfluidout.enthalpy-cfluidin.enthalpy)-Qhx;

            e(3) = cfluidin.press - cfluidout.press;

            e(4) = hfluidin.mdot*sign(obj.parameters.direction(3))+...
                hfluidout.mdot*sign(obj.parameters.direction(4));

            e(5) = hfluidin.mdot*sign(obj.parameters.direction(3))*...
                (hfluidin.enthalpy-hfluidout.enthalpy)-Qhx;

            e(6) = hfluidin.press - hfluidout.press;

        obj.parameters.pressinc = cfluidin.press;%store for use in cost
         end

         function component_cost = cost(obj)
             component_cost.cost = [0;45+9*obj.parameters.UA];
             component_cost.power = [0,0,0,0,0,0,0,0,0]';
             component_cost.emissions = [0;0;0];
             if obj.parameters.pressinc > obj.parameters.pressmax
                 component_cost.physcon = 0.01*...
                 (obj.parameters.pressinc - obj.parameters.pressmax)^2;
             else
             component_cost.physcon = 0;
             end
         end

         function y = paramcheck(obj)
             if obj.parameters.UA>=0
                 y=0;
             else
                 y=1;
            %If UA is negative, skip simulation - a meaningful solution
            %to the system will not exist.
             end
         end
    end
end
```

The Thermoelectric Power Unit component (tepowerunit) was developed separately for use in TEPSS. It is included in the base package and the code is available in [30].

II. Domains

Domain: Fluid (fluid.m)

```
classdef fluid < handle

    properties
      %note that the node variables mdot (mass flow), press (pressure)
       and enthalpy (specific enthalpy) are properties. These are the
       node variables of the fluid domain.
        temp=1
        mdot=1
        press=1
        quality=1
        enthalpy=1
        fluidprop
        fluidtype
        Tsat
        statemodel
        database

    end

    methods

        function obj = fluid(comp,ratios,database,statemodel)
            %create a fluidprop object for the fluid specified
            obj.fluidtype = comp;
            obj.database = database;
            obj.fluidprop = actxserver('FluidProp.FluidProp');
            ratzero = zeros(length(ratios),1);
            ratios = [ratios',ratzero];
            %create  and store fluidprop object
            invoke(obj.fluidprop, 'SetFluid_M', database,
                        size(ratios,1), comp, ratios);
            %switch to SI units in accordance with TEPSS' conventions
            obj.fluidprop.SetUnits('SI', '', '','');
            obj.statemodel = statemodel;
        end

        %generic property lookup
        function [property,error] = getprop(obj, propname, model,
                        state1, state2) [property,error] =
                              invoke(obj.fluidprop, propname, model,
                               state1, state2);
        end

        function allprops=allprops(obj, model, state1, state2)
%look up all available thermodynamic properties and store them in a
%structure
                    [P, T, v, d, h, s, u, q, x, y, cv, cp, c,...
                     alpha, beta, chi, fi, ksi, psi, zeta, theta, kappa,
                      gamma,eta, lambda, ErrorMsg]=
            obj.fluidprop.AllProps_M(model,state1,state2,[0,0],[0,0]);
```

```
            allprops.P = P;
            allprops.T = T;
            allprops.v = v;
            allprops.d = d;
            allprops.s = s;
            allprops.u = u;
            allprops.q = q;
            allprops.cp = cp;
            allprops.x = x;
            allprops.y = y;
            allprops.cv = cv;
            allprops.c = c;
            allprops.alpha = alpha;
            allprops.beat = beta;
            allprops.chi = chi;
            allprops.fi = fi;
            allprops.ksi = ksi;
            allprops.psi = psi;
            allprops.zeta = zeta;
            allprops.theta = theta;
            allprops.kappa = kappa;
            allprops.gamma = gamma;
            allprops.error = ErrorMsg;
end

function allpropssat=allpropssat(obj, model, state1, state2)
    %look up all thermodynamic properties and some saturation
    %properties and store them in a structure

          [P, T, v, d, h, s, u, q, x, y, cv, cp, c, alpha, beta,
           chi, fi, ksi, psi, zeta, theta, kappa, gamma, eta,
           lambda, d_liq, d_vap, h_liq, h_vap, T_sat, dd_liq_dP,
           dd_vap_dP, dh_liq_dP, dh_vap_dP, dT_sat_dT, ErrorMsg]
           =obj.fluidprop.AllPropsSat_M(model,state1,state2,
           [0,0],[0,0]);

            allpropssat.P = P;
            allpropssat.T = T;
            allpropssat.v = v;
            allpropssat.d = d;
            allpropssat.s = s;
            allpropssat.u = u;
            allpropssat.q = q;
            allpropssat.cp = cp;
            allpropssat.x = x;
            allpropssat.y = y;
            allpropssat.cv = cv;
            allpropssat.c = c;
            allpropssat.alpha = alpha;
            allpropssat.beat = beta;
            allpropssat.chi = chi;
            allpropssat.fi = fi;
            allpropssat.ksi = ksi;
            allpropssat.psi = psi;
            allpropssat.zeta = zeta;
```

```
                    allpropssat.theta = theta;
                    allpropssat.kappa = kappa;
                    allpropssat.gamma = gamma;
                    allpropssat.eta = eta;
                    allpropssat.lambda = lambda;
                    allpropssat.d_liq = d_liq;
                    allpropssat.d_vap = d_vap;
                    allpropssat.h_liq = h_liq;
                    allpropssat.h_vap = h_vap;
                    allpropssat.Tsat = T_sat;
                    obj.Tsat = T_sat;%store as a property for later
                                          calculations
                    allpropssat.dd_liq_dP = dd_liq_dP;
                    allpropssat.dd_vap_dP = dd_vap_dP;
                    allpropssat.dh_liq_dP = dh_liq_dP;
                    allpropssat.dh_vap_dP = dh_vap_dP;
                    allpropssat.dT_sat_dT = dT_sat_dT;

        end

        function initial_update(obj, guess)
            %store user supplied information about this node
           if strcmp(obj.statemodel, 'PT') == 1
               if     guess(2) == 1
                        obj.mdot = guess(3);
                     elseif guess(2) == 2
                        obj.temp = guess(3);
                     elseif guess(2) == 3
                        obj.press = guess(3);
                     else
                        disp('error, x(n,2) out of bounds, define
                                    placement in obj.update')
               end
           elseif strcmp(obj.statemodel, 'Pq') == 1
               if     guess(2) == 1
                        obj.mdot = guess(3);
                     elseif guess(2) == 2
                        obj.quality = guess(3);
                     elseif guess(2) == 3
                        obj.press = guess(3);
                     else
                        disp('error, x(n,2) out of bounds, define
                                    placement in obj.update')
               end

           else
            disp('second fluid node definition argument must be either
                        PT or Pq')
            return
           end

        end

        function lookup(obj)
            %look up thermodynamic properties of the fluid based on
user
```

```
        %supplied information
        if strcmp(obj.statemodel , 'PT') ==1
            [obj.enthalpy,error1] = obj.getprop('Enthalpy', 'PT',
                                        obj.press, obj.temp);
            [obj.quality, err] = obj.getprop('VaporQual', 'PT',
                                        obj.press, obj.temp);
        elseif strcmp(obj.statemodel, 'Pq') ==1
            obj.temp = obj.getprop('Temperature', 'Pq', obj.press,
                                        obj.quality);
            obj.enthalpy = obj.getprop('Enthalpy','Pq',obj.press,
                                              obj.quality);
        else
            disp()
            return
        end

    end

    function update(obj, x) % in setup.m bcmap and xguess colums 3
                    %require the user to supply a property #
                    %to each guess. That guess is interpreted
                    %here and assigned to the appropriate
                    %node variable.


                if     x(2) == 1
                    obj.mdot = x(3);
                elseif x(2) == 2
                    obj.enthalpy = x(3); %look up new temperature
                    [obj.temp, error2] =
        obj.getprop('Temperature', 'Ph', obj.press, obj.enthalpy);
                elseif x(2) == 3
                    obj.press = x(3); %look up new temperature
                    [obj.temp, error2] =
        obj.getprop('Temperature', 'Ph', obj.press, obj.enthalpy);
                else
                    disp('error, x(n,2) out of bounds, define
                                placement in obj.update')
                end

                if obj.temp < obj.Tsat
                    obj.quality = 0;
                elseif obj.temp>obj.Tsat
                    obj.quality = 1;
                else
                    %look up new vapor quality
                  [obj.quality, error3] = obj.getprop('VaporQual',
                                'Ph', obj.press, obj.enthalpy);
                  if strcmp(error3, 'No errors') == 0
                     disp(error3)
                     disp(obj.quality)
                  end
                end
    end

    function num = numstates(obj)
```

```
      %declare the number of node variables at each node in this domain
            num = 3;
        end
    end
end
```

A similar domain fluidconst (not shown, fluidconst.m) is used in Chapter 5 for constant specific heat applications. It receives the additional input *cp* into the constructor, which is stored as the specific heat and used for enthalpy calculations.

Domain: Mechanical Rotational (mechrot.m)

```
classdef mechrot < handle
    properties
        torque=1
        angvel=1
    end

    methods
        function update(obj, x) % in setup.m bcmap and xguess colums 3
                                  %require the user to supply a property #
                                 %to each guess. That guess is interpreted
                                   %here and assigned to the appropriate
                                   %node variable.
                    if     x(2) == 1
                        obj.torque = x(3);
                    elseif x(2) == 2
                        obj.angvel = x(3);
                    else
                        disp('error, x(n,2) out of bounds, define
                             placement in obj.update')
                    end
        end
        function num = numstates(obj)
            num = 2;
        end
    end
end
```

Domain: Electrical (electrical.m)

```
classdef electrical < handle
    %ELECTRICAL Summary of this class goes here
    %   Detailed explanation goes here

    properties
        current=1
        voltage=1
   end
```

```
    methods
        function update(obj, x, cnmap)%in bcmap and xguess
                                 %require the user to supply a property #
                                   %to each guess. That guess is
                                   %interpreted
                                   %here and assigned to the appropriate
                                   %node variable.


                        if     x(2) == 1
                            obj.current = x(3);
                        elseif x(2) == 2
                            obj.voltage = x(3);
                        else
                            disp('error, x(n,2) out of bounds, define
                                 placement in obj.update')
                        end
        end

        function num = numstates(obj)
            num = 2;
        end
    end

end
```